TAMPEREEN TEKNILLINEN YLIOPISTO

ANTTI KOLU

TYÖKONEIDEN OHJAUSJÄRJESTELMIEN AUTOMAATTISEN TESTAUKSEN KONSEPTIN JA HALLINNOINNIN KEHITYS

Diplomityö

Tarkastajat: professori Hannu Koivisto, professori Petteri Multanen
Tarkastajat ja aihe hyväksytty Automaatio-, kone- ja materiaalitekniikan tiedekunta-neuvoston kokouksessa
4. marraskuuta 2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
KOLU, ANTTI: Työkoneiden ohjausjärjestelmien automaattisen testauksen konseptin ja hallinnoinnin kehitys
Diplomityö, 58 sivua, 9 liitesivua
Huhtikuu 2010
Pääaine: Automaatio- ja informaatioverkot
Tarkastajat: professori Hannu Koivisto, professori Petteri Multanen
Avainsanat: Ohjausjärjestelmätestaus, liikkuvat työkoneet, automaattinen testaus, HIL-testaus

Liikkuvien työkoneiden ohjausjärjestelmien monimutkaistuminen ja kehittyminen on asettanut uusia vaatimuksia ohjausjärjestelmien testaamiselle. Manuaalisella testauksella ei ole mahdollista testata tehokkaasti monimutkaisia ohjausjärjestelmiä, koska manuaalisella testauksella saatava tarkkuus ja toistettavuus eivät ole riittäviä. Lisäksi monimutkaisten ohjausjärjestelmien testauksessa tarvitaan enemmän testejä, joka taas lisää testaukseen kuluvaa aikaa. Tämän takia liikkuvien työkoneiden valmistajat ja ohjausjärjestelmien valmistajat ovat ryhtyneet kehittämään simulaattoriavusteista ohjausjärjestelmien automaattista testausta. Kaupalliset toimijat tarjoavat jo HIL-testausjärjestelmiä, jotka mahdollistavat ohjausjärjestelmien automaattisen testauksen, mutta ne ovat monelta osin rajoittuneita eivätkä ota huomioon kehittyneempiä menetelmiä liittyen testien generoitiin ja datan analysointiin.

Tässä työssä on tutkittu automaattisen testauksen konseptia sekä hallinnointia. Tutkimusmateriaalina käytettiin testaukseen liittyvää kirjallisuutta ja julkaisuja, joissa esitettyjä menetelmiä sovellettiin ohjausjärjestelmien testaukseen. Lisäksi perehdyttiin jo olemassa oleviin kaupallisiin testijärjestelmiin ja niiden tarjoamiin mahdollisuuksiin.

Työ on jaettu kolmeen osaan. Ensimmäisessä osassa tutkitaan testausprosessia ja automaattisen testauksen vaiheita kuten testitapausten luontia, signaalien generointia, datan keräystä ja analysointia. Toisessa osassa esitellään automaattisen testauksen konsepti ja perehdytään tarkemmin testauksen hallinnoinnista vastaavaan ohjelmaan. Kolmannessa osassa tutkitaan automaattisen testauksen konseptin ja menetelmien toimivuutta Avant-pienkuormaajan ohjausjärjestelmän testauksessa.

Työn tuloksena huomattiin että automaattinen testausjärjestelmä mahdollistaa ohjausjärjestelmien helpomman ja laajemman testaamisen. Signaalien tarkalla automaattisella generoinnilla pystyttiin luomaan testitapauksia, joita manuaalisella testauksella on erittäin vaikea toteuttaa. Huomattiin myös, että itse testauksen automatisointi on jo melko kehittynyttä, jonka takia automaattista testausta tulisi tulevaisuudessa kehittää testien generoinnin ja tulosten analysoinnin osa-alueilla. Tämä mahdollistaisi testauksen nopeamman läpiviennin kehitysprosessin aikaisemmissa vaiheissa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
KOLU, ANTTI: Development of automatic testing concept and administration in mobile machine control system testing
Master of Science Thesis, 58 pages, 9 Appendix pages
April 2010
Major: Automation- and information networks
Examiners: Professor Hannu Koivisto, Professor Petteri Multanen
Keywords: Control system testing, mobile machines, automatic testing, HIL-testing

The control systems in mobile work machines have become more complex in recent years and that has created new requirements for control system testing. The precision and repeatability that can be achieved with manual testing is not enough for these complex control systems. Also the number of tests and the time spent running those increase when the complexity of control system increases. This is why machine manufacturers and control system manufacturers have started to develop automatic testing of control systems. There are few commercial systems available that promise to enable automatic testing of control systems. However these systems are not yet fully developed and they have constraints in some areas of testing.

The research material used for studying automatic testing of control systems were taken mostly from software testing literature. Software testing methods and means were then adapted for control system testing. Also the commercially available systems were reviewed for their potential.

This thesis is separated to three parts. Testing processes and the phases in automatic testing are examined in the first part. There we examine test case generation, signal generation, data retrieval and data analyzing. In the second part the automatic testing concept and a program used for controlling the testing process are introduced. In the third part automatic testing concept is used for testing Avant multipurpose loader.

As a result it was noticed that automatic testing enables easier and more extensive testing of control systems. Precise signal generation enables to create tests that are very difficult or even impossible to do with manual testing. It was also noticed that the automation of test processes is quite advanced which is why in future automatic testing should be developed in the areas of test case generation and analyzing. This would allow faster test cycles in the earlier phases of development process.

ALKUSANAT

Tämä työ on tehty Tampereen teknillisessä yliopistossa systeemitekniikan automaatio- ja informaatioverkkojen pääaineeseen hydrauliikan ja automatiikan laitoksella. Työn tavoitteena oli tutkia työkoneiden ohjausjärjestelmien automaattista testausta sekä esitellä ja testata automaattisen testijärjestelmän toimintaa.

Haluan kiittää työni tarkastamisesta ja neuvoista professori Hannu Koivistoa sekä professori Petteri Multasta. Työni on tehty osana TINAT- ja GIM-projekteja ja haluankin kiittää projekteissa mukana olleita ja erityisesti työkavereitani Petri Manteretta, Arvi Vallasta ja Riku Varista, joiden kanssa olen tehnyt läheistä yhteistyötä testausmenetelmien kehityksessä ja tutkimuksessa. Lisäksi haluan kiittää TkT Mika Hyvöstä hänen avustaan Avant-simulaattorin käytössä ja testaukseen liittyvistä ideoista.

Kiitokset myös vanhemmilleni, jotka ovat tukeneet minua opintojen ja diplomityön tekemisen aikana. Lopuksi vielä haluan esittää kiitokseni kaikille ystävilleni ja läheisilleni sekä opiskelukavereilleni Tampereella mukavasta sekä innostavasta opiskeluilmapiiristä.

18.3.2010

Antti Kolu

SISÄLLYS

LYHENTEET JA MÄÄRITELMÄT

ASAM	Association for Standardization of Automation and Measuring Systems
ASAM AE HIL	Standardisoitu HIL- rajapinta
Black-box testing	Testaustapa, jossa testattavan järjestelmän sisäisiä muuttujia ja toimintaa ei tunneta
ECU	Electronic Control Unit
FIMA	Forum for Intelligent Machines
Functional testing	Testaustapa, jossa testataan järjestelmän ulostuloja suhteessa sisäänmenoihin
GIM	Generic Intelligent Machines
GIMsim Sequence Generator	Testiesekvenssejä ajava ohjelma
GIMsim Test Analyzer	Testejä analysoiva ohjelma
GIMsim Test Manager	Testejä suorittava ohjelma
HIL	Hardware In the Loop
IHA	Department of Intelligent Hydraulics and Automation
MASI	Tekesin mallinnus ja simulointi-ohjelma
MIL	Model In the Loop
NI	National Instruments
PIL	Processor In the Loop
SIL	Software In the Loop
Tekes	Teknologian ja innovaatioiden kehittämiskeskus
TestLink	Vapaan lähdekoodin testauksen hallinta ohjelma
TINAT	Työkoneen integrointivaiheen automaattinen testaus
TTY	Tampereen teknillinen yliopisto
UML	Unified Modeling Language
White-box testing	Testaustapa, jossa hyödynnetään testattavan järjestelmän sisäistä tuntemusta

1. JOHDANTO

Testaus on tärkeä osa laadunvarmistusta kaikessa teollisessa tuotannossa. Sitä käytetään kaikissa tuotteen kehitysprosessin vaiheissa määrittelystä lopputuotantoon. Erilaisille tuotteille ja eri kehityksen vaiheissa oleville tuotteille tarvitaan erilaisia testaustekniikoita ja menetelmiä.

Ohjausjärjestelmiä on nykyään joka puolella. Autot, lentokoneet, metsäkoneet yms. toimivat paljolti ohjausjärjestelmien varassa ja monet näiden laitteiden uudet ominaisuudet on kehitetty ohjausjärjestelmiä kehittämällä. Tämä on kuitenkin johtanut ohjausjärjestelmien kasvamiseen ja monimutkaistumiseen. Samalla myös ohjausjärjestelmien testauksen tarve on kasvanut.

FIMA ry on työkonevalmistajien ja osavalmistajien yhteistyöfoorumi, jonka avulla koordinoidaan alan kehitystä [1]. Foorumin yhtenä tavoitteena on nimenomaan liikkuvien työkoneiden, kuten metsätyökoneiden ja kauhakuormaajien, ohjausjärjestelmätestauksen kehittämien. Tekesin mallinnus ja simulointi-ohjelman TINAT-projektissa pyritään kehittämään liikkuvien työkoneiden integraatiovaiheen ohjausjärjestelmätestauksen automatisointia. FIMA on ollut tässä projektissa osarahoittajana. Tämä diplomityö on tehty osana älykkäiden koneiden huippuyksikkö hanketta (GIM) ja TINAT-projektia.

Tämän työn tarkoituksena on perehtyä ohjausjärjestelmien kehitys- ja testausprosesseihin sekä testien generointimenetelmiin, joita voidaan käyttää automaattisessa ohjausjärjestelmätestauksessa. Lisäksi perehdytään automaattisen ohjausjärjestelmätestauksen testijärjestelmälle asettamiin vaatimuksiin sekä esitellään testien automaattiseen suoritukseen kehitetty ohjelma.

Työn alkuvaiheessa pyrin etsimään kirjoja ja julkaisuja, jotka liittyivät ohjausjärjestelmien automaattiseen testaukseen. Hyvin pian tuli kuitenkin selväksi ettei aihetta oltu tutkittu paljoa ja hyvien lähteiden löytäminen oli hankalaa. Ohjausjärjestelmätestauksesta ei ole kirjoitettu kovin montaa kirjaa ja näistäkin vain harvoissa on otettu huomioon testauksen automatisointia. Tämän takia jouduin hakemaan tietoa muista testaukseen liittyvistä kirjoista ja soveltamaan niitä ohjausjärjestelmätestaukseen. Esimerkiksi ohjelmistotuotannon testausmenetelmistä löytyi testitapausten generointiin paljon tietoa. Julkaisujen suhteen tilanne on samankaltainen. Julkaisuja tosin löytyi, mutta ne käsittelevät paljolti suurten yritysten ratkaisuja ilman varsinaista tutkimuksellista tai teoreettista osaa.

Luvussa kaksi perehdytään nykyään käytössä oleviin ohjausjärjestelmien testausjärjestelmiin. Siinä esitellään erilaisia kehitys- ja testausprosessin malleja, joiden avulla voidaan parantaa kehityksen laatua ja nopeutta. Erilaisia testitapausten generointimenetelmiä käydään läpi, joita voitaisiin käyttää hyväksi automaattisessa testien generoinnissa. Lisäksi kappaleen lopussa on lyhyet esittelyt signaalien generoinnista testausjärjestelmissä ja testauksessa saadun datan analysoinnista.

Luvussa kolme esitellään automaattisen testauksen konsepti, joka kehitettiin osana edellä mainittuja projekteja. Se kuvaa tarvittavaa testausjärjestelmää jaettuna viiteen osaan. Nämä osat ovat testien hallinnointi, testauksen hallinnointi, testauksen suoritus, reaaliaikainen simulointi ja analysointi. Samassa luvussa esitellään konseptin toteuttava testausjärjestelmä, joka on käytössä Tampereen teknillisen yliopiston (TTY) hydrauliikan ja automatiikan laitoksella (IHA).

Luvussa neljä esitellään GIMsim Test Manager-ohjelmiston kehitystyötä. Siinä kuvataan miten Matlab-ympäristöön on mahdollista toteuttaa testauksen hallinnoinnista vastaava ohjelma. Sen toiminnassa on pyritty vastaamaan luvussa kaksi esiteltyihin menetelmiin, joilla automaattinen testaus on mahdollista toteuttaa. Ohjelman toiminta ja rakenne on kuvattu erillisissä aliluvuissa.

Luvussa viisi esitellään tuloksia, joita on saatu testausjärjestelmän integroinnissa Avant-työkoneen HIL-simulaattoriin. Avant-työkoneen simulaattori on kehitetty IHA:ssa GIM-projektin tarpeisiin, mutta se sopii ohjausjärjestelmän puolesta myös ohjausjärjestelmän automaattisen testauksen tutkimukseen. Konseptia testattiin ohjausjärjestelmän osalla, joka vastaa dieselmoottorin ohjauksesta. Saatuja tuloksia ja jatkokehitysideoita on käyty läpi luvun lopussa. Luvussa kuusi esitellään työn loppuyhteenveto. Siinä esitellään työssä saadut tulokset ja pohditaan parannuksia sekä mahdollisia jatkokehitys-suuntia.

2. OHJAUSJÄRJESTELMIEN DYNAAMINEN TESTAUS

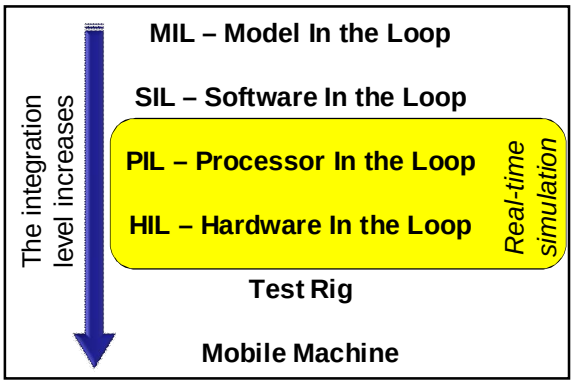
Testaus on yksi osa automaatiojärjestelmän elinkaarta. Testausta suoritetaan eri vaiheissa ja eri tavalla, mutta sen tavoitteena on aina parantaa lopputuotteen luotettavuutta. Automaation luotettavuutta lisäävät tekijät voidaan jakaa neljään osaan, jotka ovat virheiden välttäminen, vikasietoisuuden lisääminen, vikojen löytäminen ja ennustaminen. Virheiden välttäminen ja ennustaminen ovat molemmat inhimillisiä asioita ja niitä saa kehitettyä parhaiten kokemuksen avulla. Vikasietoisuuden lisäämiseen tarkoittaa järjestelmän rakenteen muokkaamista niin, että vikojen vaikutusta saadaan rajattua. Esimerkiksi sähkö- ja viestilinjojen kahdentamien on yksi vikasietoisuutta parantava tekijä. Vikojen löytäminen tarkoittaa käytännössä testaamista.

Yksi testaamisen menetelmä on dynaaminen testaus, johon tässä luvussa keskitytään. Dynaaminen testaus on yksi tärkeimmistä ja yleisimmin käytetyistä ohjelmistojen laadunvarmennus menetelmistä. Se on käytännössä ainoa testaustapa, jolla voidaan ottaa riittävästi huomioon sekä kehitys- että operointiympäristö ja ohjelmistojen vaatimukset. Dynaamisella testauksella tarkoitetaan yleisesti ohjelmiston tai järjestelmän testaamista tietyillä sisääntulon arvoilla sekä tutkimalla, että niistä johtuvat ulostulojen arvot ovat odotettujen kaltaisia. Kirjassa *automotive embedded system handbook* on määritelty dynaaminen testaus kolmella kriteerillä. Testattavan järjestelmän tulee reagoida aidosti, muuttuviin sisääntulojen arvoihin. Testattava järjestelmää testataan joko aidossa tai simuloidussa ympäristössä. Testattava järjestelmä suoritetaan askeleittain eli diskreetisti.[2]

Ohjausjärjestelmät ovat automaatiojärjestelmiä, jotka ohjaavat muita järjestelmiä ja laitteita. Ne ottavat sisääntuloina mittaus ja ohjausdataa, joiden perusteella lasketaan ulostulot. Ulostulot toimivat muiden järjestelmien ohjauksina. Ohjausjärjestelmiä on lähes kaikissa sähköisesti ohjatuissa koneissa radio-ohjattavista lentokoneisiin ja ne koostuvat laitteistosta ja ohjelmistosta. Liikkuvissa työkoneissa laitteistona toimivat usein ohjausyksiköt ja CAN-väylät, joita pitkin kulkee mittaus ja ohjausdata ohjausyksikön ja muun järjestelmän välillä. Ohjelmisto määrää miten ohjausjärjestelmä toimii. Sen perusteella lasketaan ohjausyksiköstä ulostulevat ohjaussignaalit.

Hardware In the Loop-testaus (HIL) on yksi dynaamisen testauksen menetelmä. Siinä ohjausjärjestelmä liitetään kiinni simulaattoriin, joka simuloi laitteiston toimintaa. Ohjausjärjestelmä siis luulee olevansa kiinnitettynä oikeaan ohjattavaan laitteistoon. Tämä mahdollistaa ohjausjärjestelmän testaamisen ennen itse laitteiston kehittämistä. HIL-simulaattoreita ja testauslaitteistoja on yleisesti käytössä työkoneita valmistavissa

yrityksissä sekä autoteollisuudessa. Kuvassa 1 on esitetty työkoneiden testauksessa käytettäviä menetelmiä järjestelmän eri kehitysvaiheissa.



Kuva 1. Testaustavat työkoneiden eri kehitysvaiheissa. [3]

HIL- testauksen lisäksi Prosessor In the Loop (PIL), Software In the Loop (SIL) ja Model In the Loop (MIL) testausta käytetään erityisesti kehityksen aikaisemmissa vaiheissa. Niiden avulla on helpompaa varmentaa menetelmien ja funktioiden toimivuus ilman oikeaa laitteistoa. Tässä työssä keskitytään kuitenkin reaaliaikasimulointia sisältävään testaukseen ja erityisesti HIL osioon.

Tässä luvussa perehdytään työkoneiden ohjausjärjestelmien dynaamiseen testaukseen ja kuvataan eri vaiheita sekä asioita, joita testauksessa ja sen valmistelussa tulee ottaa huomioon. Luvussa keskitytään erityisesti testausprosessiin ja testitapausten luomiseen ohjausjärjestelmien tapauksessa.

Ensimmäiseksi perehdytään nykyisiin ohjausjärjestelmien testausjärjestelmiin. Seuraavaksi käydään läpi erilaisia ohjausjärjestelmän kehitys- ja testausprosesseja. Tämän jälkeen lähdetään käymään läpi testauksen eri vaiheita prosessin mukaisessa järjestyksessä.

2.1. Ohjausjärjestelmien testaus nykyään

Liikkuvien työkoneiden ohjausjärjestelmien testauksen toteutus vaihtelee hyvin paljon yritysten ja testauslaitteistojen välillä. Simulointi- ja testausjärjestelmiä pyritään standardisoimaan, eurooppalaisen autoteollisuuden johdolla mm. ASAM-standardisointijärjestössä. ASAM, Association for Standardisation of Automation and Measuring Systems, perustettiin saksalaisten autonvalmistajien toimesta 1998, jonka jälkeen siihen on liittynyt myös muita auto- ja laitevalmistajia sekä ohjelmistokehitysyrityksiä. [4]

Teollisuudenaloista auto- ja lentokoneteollisuudessa ollaan tällä hetkellä pisimmällä ohjausjärjestelmien testauksen kehityksessä. Yllämainittujen yritysten laitteet ovatkin pääosin suunnattu kyseisten teollisuusalojen käyttöön. Autoteollisuus on alana lähempänä työkoneita, joten joitain siellä käytettyjä menetelmiä ja laitteita voidaan soveltaa osittain myös työkoneiden testaukseen. [5]

Suurilla simulointiin ja ohjausjärjestelmiin keskittyvillä yrityksillä on omia järjestelmiä, joilla pystytään toteuttamaan HIL-testausta ja joissain tapauksissa myös

automatisoitua testausta. Kyseisiä yrityksiä ovat esimerkiksi dSpace, National Instruments (NI) ja MathWorks. Projektin aikana suoritettujen kokeilujen ja tiedonhankinnan perusteella dSpace näyttäisi olevan yrityksistä pisimmällä automaattisen testauksen saralla. [5,6]

DSpace tarjoaa sekä HIL-simulaattoreita että ohjelmistoja, joiden avulla voidaan suorittaa manuaalista ja automaattista testausta. Automaattisen testauksen ohjelmistona käytetään tällöin AutomationDesk-ohjelmaa. Siinä luodaan testejä joissa ladataan simulointimallit ja alustetaan ne suunnitellulla tavalla. Signaalien luonti tapahtuu valmiilla funktioilla, joilla voidaan luoda erimuotoisia signaaleja, kuten ramppeja ja sinisignaaleja. Signaaleja voidaan tallentaa ja analysoida jälkikäteen. Simuloinnissa käytetään Simulink-malleja, joten myös MathWorksin Simulink tuotteita tarvitaan.

MathWorksin Simulink-malleja käytetään suurimmassa osassa HIL-simulaattoreita, koska ne ovat hyvin toimivia ja yhteensopivia erilaisten ohjelmistojen kanssa. MathWorks ei tosin tarjoa omia ohjelmistoja ohjausjärjestelmätestaukseen, joten tällöin joudutaan ohjelmat kehittämään itse tai käyttämään kolmannen osapuolen sovelluksia. MathWorksin HIL-simulaattori on nimeltään xPC Target, johon on tarjolla API rajapintoja, joiden kautta testausta hallitaan.

National Instruments tarjoaa myös HIL-simulaattoreita ja ohjelmistoja ohjausjärjestelmätestaukseen. Heidän kehittämällään LabVIEW-ohjelmointikielellä voidaan rakentaa reaaliaikasimulaattoreissa pyöriviä simulaatiomalleja, mutta ne toimivat ainoastaan National Instrumentsin omissa HIL-simulaattoreissa. Lisäksi kokemusten perusteella monimutkaisten simulaattorien rakentaminen LabVIEW-kielellä on vaikeaa. National Instrumentsin HIL-simulaattoreissa voidaan käyttää myös Simulink-malleja, mutta tällöin törmätään usein yhteensopivuusongelmiin ja esimerkiksi kaikkia parametreja ei ole mahdollista muokata eikä kaikkia signaaleja ole mahdollista tallentaa.

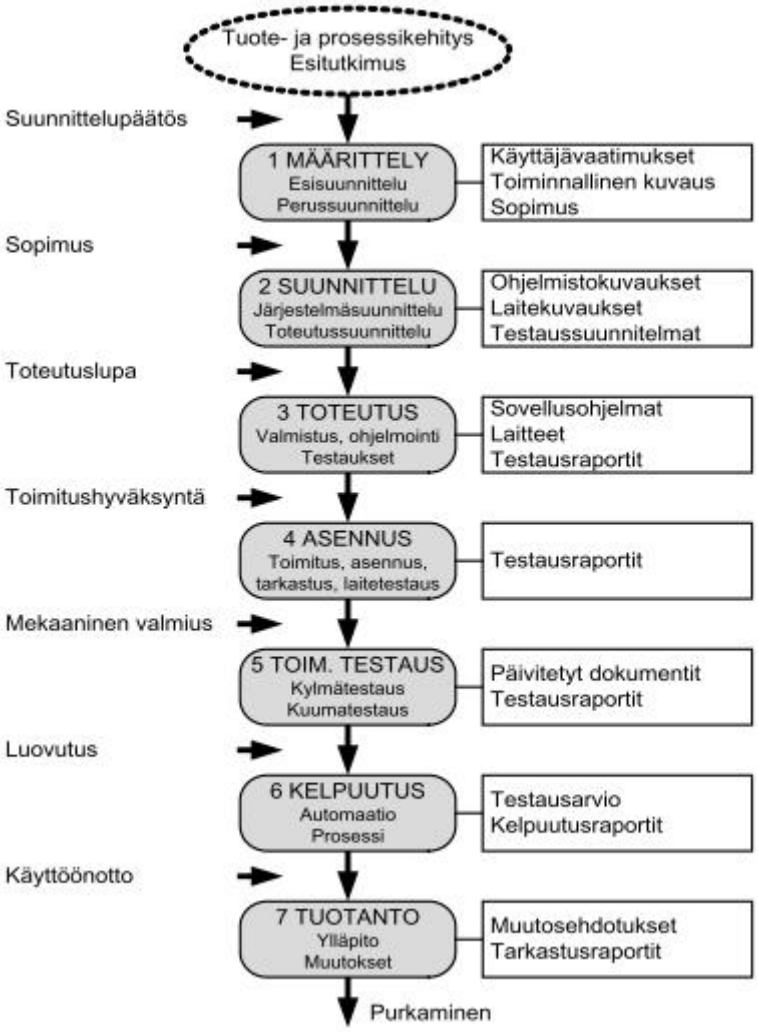
Automaattisen testauksen ohjelmana käytetään National Instrumentsillä TestStand-ohjelmaa. Siinä on mahdollista kehittää ja suorittaa testejä AutomationDeskin kaltaisesti.

Yleisesti voidaan sanoa, että laitteistoja ja ohjelmistoja ohjausjärjestelmien automaattiseen testaukseen on tarjolla, mutta niiden ominaisuudet rajoittuvat vielä testauksen suorittamiseen. Ohjelmat eivät tarjoa kehittyneitä menetelmiä, kuten testien automaattista generointia tai analysointia, joilla testausta saadaan tehostettua. Automation Deskiin on mahdollista rakentaa omia lisäosia python-ohjelmointikielellä, joilla testausta saadaan kehitettyä omien tarpeiden mukaan. LabVIEW:ssa ei ole samanlaista laajennettavuutta, joten omien testausmenetelmien kehittäminen on tällöin vaikeampaa.

2.2. Kehitys- ja testausprosessi

Ohjausjärjestelmien kehitysprosessia voidaan kuvata elinkaarimallilla, josta selviää järjestelmän läpikäymät vaiheet sen kehityksen aloittamisesta aina kehityksen ja tuen

lopettamiseen asti. Kuvassa 2 on esitetty automaatiojärjestelmän elinkaaren vaiheet, johon ohjausjärjestelmien kehitys voidaan samaistaa. Elinkaarimallissa kuvataan vaiheiden lisäksi myös tarvittavia ja tuotettavia dokumentteja sekä välivaiheita.



Kuva 2. Automaatiojärjestelmän elinkaaren vaiheet, niiden väliset etapit sekä tulokset. [7]

Elinkaaren kuvaus voidaan aloittaa kehityksen ja tuotekehityksen perusteella tehdystä suunnittelupäätöksestä. Tällöin aloitetaan ohjausjärjestelmän määrittely. Tässä kohdassa kirjoitetaan ohjausjärjestelmän toiminnallinen kuvaus ja kuvataan sille asetetut käyttäjävaatimukset. Näiden määrittelyjen tulee olla niin tarkkoja, että niiden perusteella voidaan kirjoittaa sopimus ohjausjärjestelmän kehityksestä. [7]

Suunnittelu aloitetaan, kun sopimus on tehty. Tällöin tarkennetaan määrittelyvaiheessa tehtyä suunnittelua ohjelmisto- ja laitekuvauksilla sekä testaussuunnitelmalla. Suunnittelun lopuksi voidaan suunnitelmaa alkaa toteuttamaan. [7]

Toteutusvaiheessa valmistetaan, kootaan ja testataan ohjausjärjestelmä. Toteutusvaiheen lopussa tuote on testattu valmis siirrettäväksi oikeaan järjestelmään. [7]

Asennusvaiheessa ohjausjärjestelmä siirretään oikeaan järjestelmään. laitteistotestauksen avulla tarkistetaan, että järjestelmä on asennettu oikein ja on

laitteistokuvausten mukainen. Asennuksen jälkeen järjestelmä on valmis toiminnalliseen testaukseen. Toiminnallisen testausvaiheen tarkoituksen on osoittaa että ohjausjärjestelmä toimii halutusti ja se voidaan luovuttaa asiakkaalle. [7]

Kelpuutusvaiheessa tarkastetaan, että järjestelmä on dokumentoitu riittävästi ja niiden perusteella voidaan todeta järjestelmän luotettavuus. Tämä on erityisen tärkeää korkeaa turvallisuustasoa vaativissa järjestelmissä kuten voimalaitoksissa ja lääketeollisuudessa, mutta myös liikkuvissa työkoneissa. Tuotantovaiheessa voidaan suorittaa pieniä korjauksia ja muutoksia, jotka lähtevät liikkeelle elinkaaren määrittelyvaiheesta. [7]

Tämän kaltainen elinkaarimalli kuvaa yleisellä tasolla automaatiojärjestelmien kehitystä. Tästä voidaan erottaa omaksi osakseen kehitys- ja testausprosessi, jossa määrätään suuntaviivat miten kehitys viedään läpi ja mitä missäkin kehitysvaiheessa testataan. Tällöin tulee erotella eri vaiheet missä testausta suoritetaan ja miten testaaminen eroaa kussakin vaiheessa. Ohjausjärjestelmien tapauksessa lisähaastetta testaukseen tuovat laitteiston ja ohjelmiston samanaikainen kehitys sekä järjestelmän hajauttaminen. Jokaisella osajärjestelmällä voi esimerkiksi olla oma kehitystiimi, joista jokainen voi toimia itsenäisesti. Tällöin on tärkeää, että testaus on hyvin organisoitua ja että tiedot löytyvät keskitetysti yhdestä paikkaa.

2.2.1. Testauksen tasot

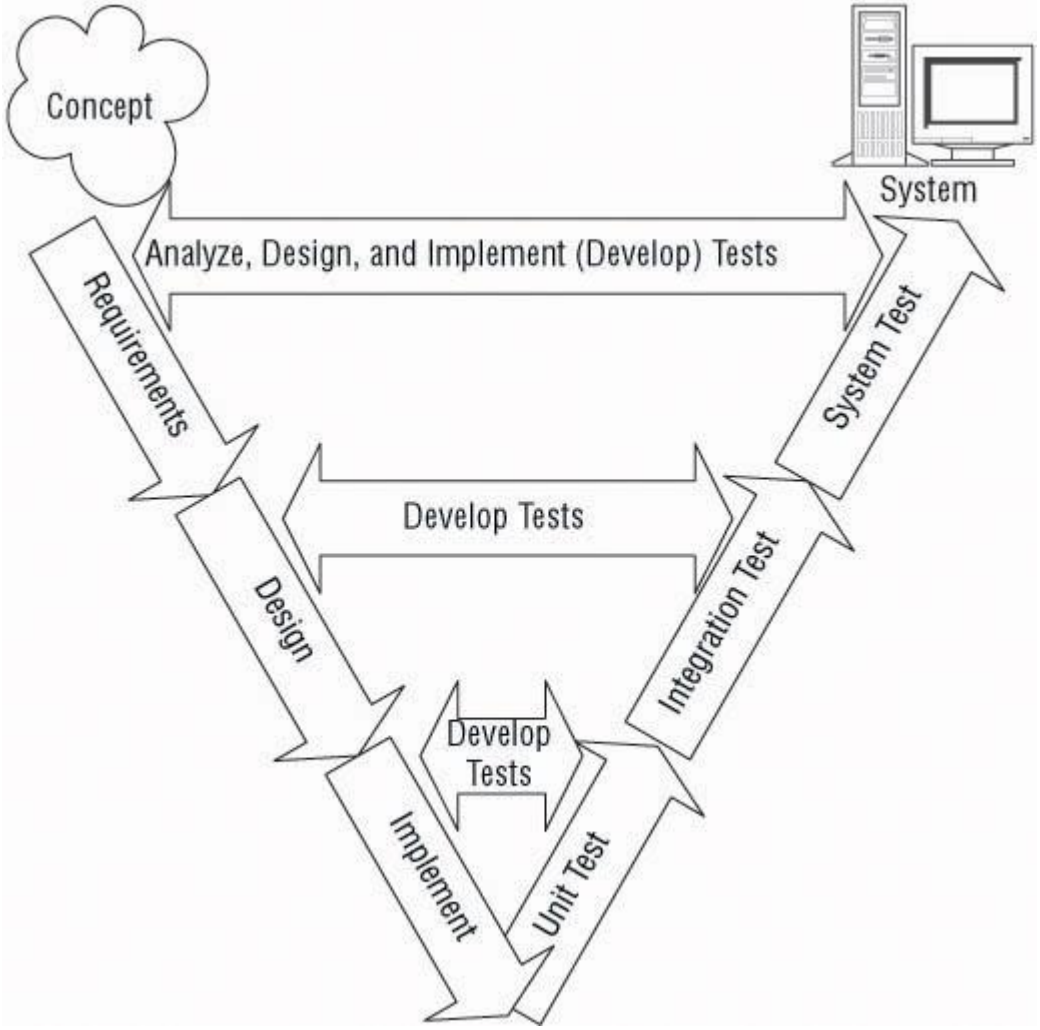
Kehitysprosessin eri vaiheissa tapahtuvaa testausta voidaan jaotella eri tasoihin sen mukaan mitä testataan ja missä vaiheessa. Kehitettäessä uutta ohjausjärjestelmää pyritään kehityksen kaikissa vaiheissa testaamaan jo kehitetty yksikkö mahdollisimman tarkasti. Yksikkötestauksella tarkoitetaan ohjausjärjestelmätestauksessa yhden ohjausyksikön testaamista. Tällöin tutkitaan ohjausyksikön sisäistä toimintaa ennen sen liittämistä isompaan kokonaisuuteen. Integraatiotestauksessa taas tutkitaan monen ohjausyksikön keskinäistä yhteistoimintaa. Kun kaikki ohjausyksiköt on liitetty samaan järjestelmään, voidaan ruveta puhumaan järjestelmätestauksesta.

Kun ohjausjärjestelmän yksi versio on saatu valmiiksi, siirrytään ns. ylläpitovaiheeseen. Tällöin ohjausjärjestelmää kehitetään edelleen, mutta testaus tapahtuu pääsääntöisesti vanhaa järjestelmää vastaan. Tätä testausmenetelmää kutsutaan regressio testaukseksi [2]. Siinä saman ohjausjärjestelmän eri versioita verrataan keskenään ja tarkastetaan, että vanhassa ohjausjärjestelmässä toimineet toiminnot toimivat uudessakin.

Tuotantolinjalla tapahtuvaa testausta nimitetään tuotantotestaukseksi ja se suoritetaan useimmiten back-to-back-testauksena. Tällöin jokaiselle tuotantolinjalta tulevalle ohjausjärjestelmälle suoritetaan joukko testejä, joiden tuloksia verrataan toimivalla ohjausjärjestelmällä suoritettuihin vastaaviin testeihin. [2]

2.2.2. V- malli

Yleisesti minkä tahansa järjestelmän elinkaaren kuvaamiseen käytetään vesiputousmallia. Se kuvaa järjestelmän suunnittelua, jossa lähdetään liikkeelle korkean tason määrittelyistä ja edetään aina pienempiin kokonaisuuksiin. Vesiputousmallista kehitettyä muunnelmaa, joka ottaa testauksen huomioon, kutsutaan V-malliksi. Se on esitettynä kuvassa 3. [8]



Kuva 3. Yleinen järjestelmäkehityksen ja testauksen V-malli. [8]

V-mallissa kuvataan kehitysprosessi vesiputoukseksi, jonka testausosa on käännetty takaisin ylöspäin. Näiden kahden puolen välissä olevat nuolet kuvaavat kehityksen iterointia ja tiedon kulkua. Vasemmalta oikealle siirtyvät tiedot, joita tarvitaan testauksen suorittamiseen ja määrittämään onko testaustulokset hyväksyttäviä. Oikealta vasemmalle taas tapahtuu iterointia, jos testauksessa havaittiin virheitä, jonka takia joudutaan palaamaan takaisin suunnitteluun.

Testauksessa löydettyjen virheiden rahallinen ja ajallinen vaikutus kasvaa sitä mukaa mitä korkeammalle ja pidemmälle testausprosessissa päästään ennen virheen löytymistä. Helpoimmalla ja halvimmalla päästään, kun virheet löytyvät jo

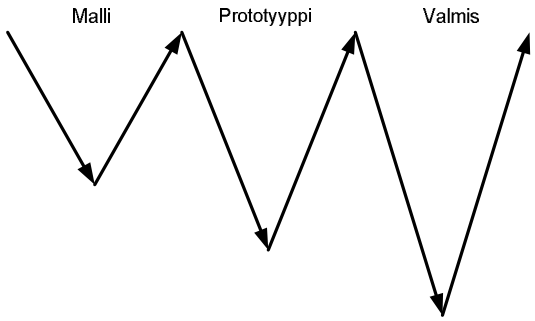
yksikkötestausvaiheessa, jolloin ainoastaan kyseistä yksikköä tarvitsee korjata. Jos virhe löytyy puolestaan prosessin lopussa, pahimmillaan kehitystyö täytyisi aloittaa alusta. [8]

Mallin huonona puolena on sen soveltuvuus ohjausjärjestelmien kehitykseen. Ohjausjärjestelmiä kehitetään usein vaiheittain niin, että ensin kehitetään mallipohjainen ratkaisu, jota ruvetaan korvaamaan oikeilla järjestelmillä sen mukaan, kun ne valmistuvat. [9]

2.2.3. Usean V:n malli

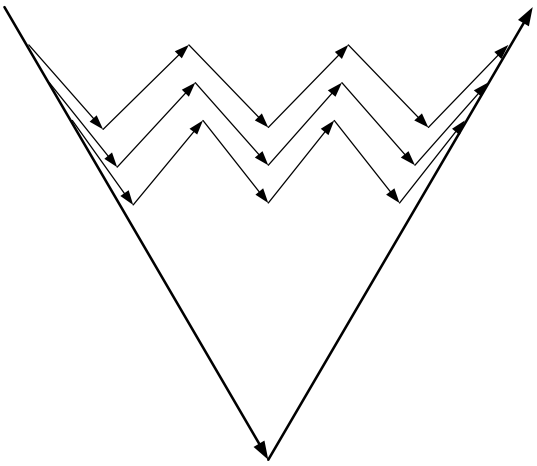
Usean V:n malli on yleisnimike malleille, joissa käytetään montaa V- mallin sykliä. Nämä voidaan jakaa karkeasti kahteen joukkoon, joissa useita V-malleja sijoitetaan peräkkäin tai päällekkäin.

Peräkkäin sijoitetuissa V-malleissa on ideana kuvata kehitysprosessissa tulevia välituloksia. Esimerkiksi ohjausjärjestelmän kehityksessä saatetaan ensin kehittää mallipohjainen järjestelmä, toiseksi prototyyppi ja vasta kolmanneksi valmis ohjausjärjestelmä. Kuvassa 4 on kuvattuna kyseinen kehitysprosessi.



Kuva 4. Peräkkäinen V-malli.

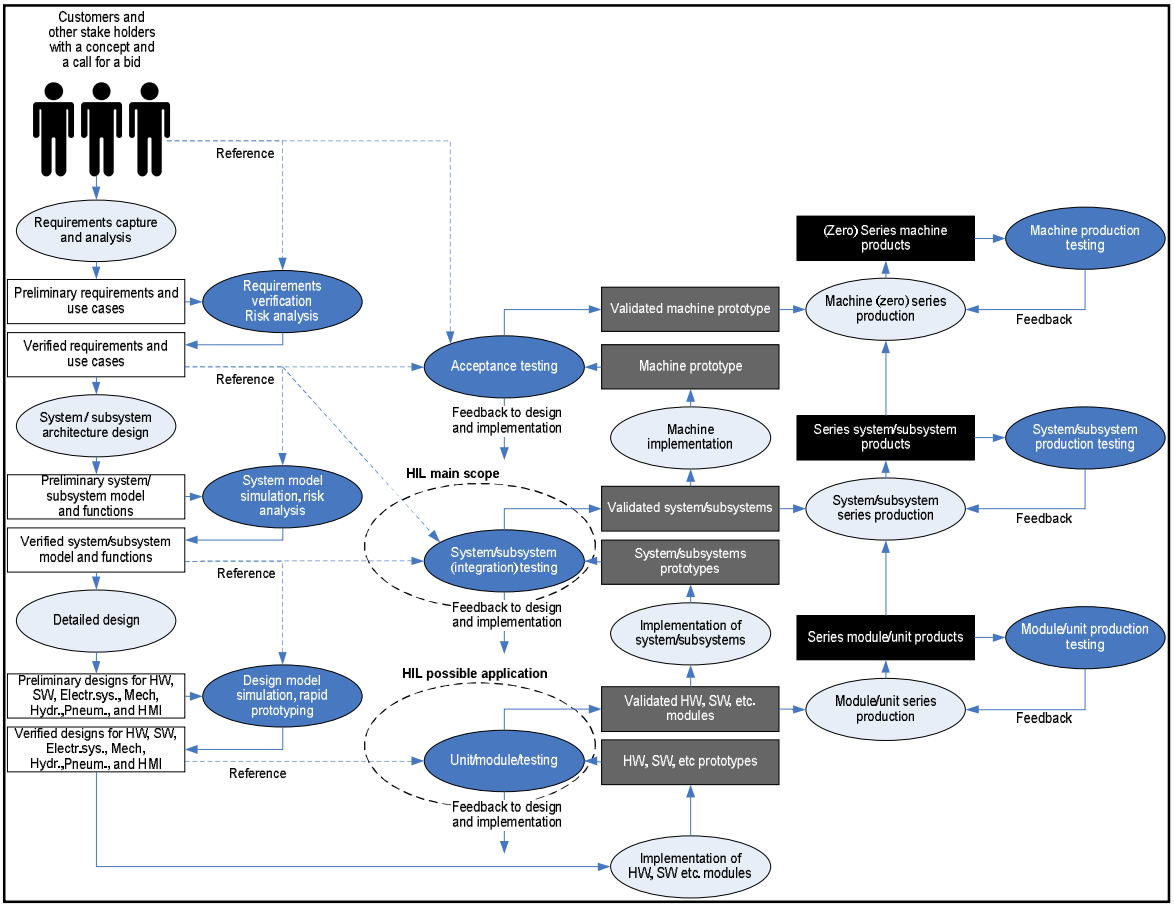
Sisäkkäin sijoitettujen V-mallien tapauksessa kuvataan komponenttien samanaikaista kehitystä. Tällöin monia eri osakomponentteja kehitetään yhtä aikaa ja prosessi voi myös sisältää peräkkäisen V-mallin. Tämä malli, joka on esitettynä kuvassa 5, kuvaa ehkä realistisimmin kehitysprosessia ja siihen liittyvää testausta.



Kuva 5. Sisäkkäinen V-malli.

2.2.4. Tripla V-malli

Usean V:n malli on myös tutkija Jarmo Alasen kehittämä tripla V-malli, joka on esitettynä kuvassa 6. Se kehitettiin mallintamaan ohjausjärjestelmän kehitysprosessia, jossa huomioidaan sekä ohjelmisto- että laitteistokehitys. Suurempi kuva kyseisestä mallista on liitteenä 1. [3]



Kuva 6. Ohjausjärjestelmän kehitysprosessin tripla V-malli. [3]

Malli lähtee liikkeelle vaatimustenmäärittelystä ylävasemmalta, josta lähdetään etenemään alaspäin kuten normaalissa V-mallissa. Vasemmassa reunassa on kolme kokonaisuutta jotka ovat vaatimustenmäärittely, järjestelmä- ja osajärjestelmä-suunnittelu sekä yksityiskohtainen suunnittelu. Jokaisessa kolmessa kohdassa suoritetaan myös testaukseen ja laadunvarmistukseen liittyviä toimintoja, kuten riskien analysointia, mallien simulointia ja rapid prototyping-prosessia.

Kun yksityiskohtainen suunnittelu on saatu valmiiksi, siirrytään järjestelmien toteutukseen ja testaukseen, joka on kuvattuna keskimmäisillä laatikoilla. Tämä vastaa V-mallin oikeaa reunaa, jossa toteutetaan suunnitellut järjestelmät ja ohjelmistot. Siinä lähdetään ensin toteuttamaan yksityiskohtaisen suunnittelun tuloksia sekä testaamaan niitä yksikkötestauksella. Kun tämä taso on valmis, siirrytään mallien ja alimallien toteutukseen sekä näiden integraatiotestaukseen, josta lopulta siirrytään prototyypin valmistukseen ja testaukseen. Näiden testausten aikana voidaan havaita virheitä joiden

takia joudutaan siirtymään edelliseen vaiheeseen. Tämä on luonnollista iterointia, jota esiintyy ohjausjärjestelmien suunnittelussa ja toteutuksessa.

Järjestelmien toteutuksessa ja testauksessa käytetyt määritelmät saadaan vasemman laidan suunnittelu osioista. Näiden määritelmien heijastumista testaukseen ja toteutukseen on kuvattu katkoviivanuolilla.

Mallin oikea laita kuvaa varsinaista tuotantovaihetta. Silloinkin tuotanto lähtee alemmista järjestelmistä, alimalleihin ja aina valmiiseen tuotteeseen asti. Näissä osioissa suoritetaan tuotantotestausta.

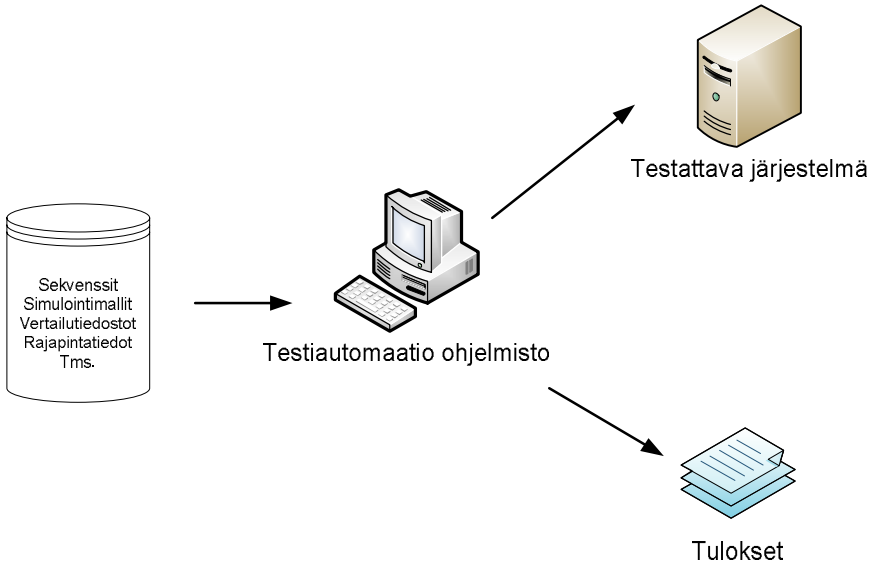
2.3. Automaattisen testijärjestelmän rakenne

Jotta kehitys- ja testausprosessin mukainen järjestelmä voidaan toteuttaa, tulee testijärjestelmän rakenteen tukea sitä. Tämä tarkoittaa, että rakenteen tulee olla mahdollisimman joustava erilaisille testaustekniikoille ja muutoksille testeissä tai järjestelmässä. Kuten luvussa 2.2.2 mainittiin, tapahtuu ohjausjärjestelmien kehitys usein vaiheittain, jolloin mallipohjaisilla ratkaisuilla testataan ensin konseptin toimivuutta ja vasta tämän jälkeen siirrytään HIL-testaukseen. MIL, SIL ja PIL testauksen aikana luodut testitapaukset tulisi siis olla mahdollista suorittaa myös HIL-testauksen aikana. Tämä on toteutettavissa, kun järjestelmän eri osia eriytetään toisistaan.

Kirjassa *testing embedded software* [9 ss. 218-222] luetellaan neljä tapaa vähentää osien riippuvuuksia toisistaan. Nämä ovat:

- Testitapaukseen liittyvä data erotetaan testitapauksen kuvauksesta
- Testitoiminnon kuvaus erotetaan sen teknisestä toteutuksesta
- Automaattisen testausprosessin toteutus eriytetään testausympäristöstä ja testattavasta järjestelmästä
- Testattavan järjestelmän ja testitapauksen välinen kommunikaatio ei saa olla osana testitapauksen suoritusta

Yllämainitun kaltainen järjestelmä on esitettynä kuvassa 7, joka on muokattu testing embedded software kirjan kuvasta [9 s. 219].



Kuva 7. Automaattisen testijärjestelmän periaatekuva, joka on muokattu testing embedded software kirjan kuvasta. [9 s. 219]

Kuvan vasemmassa reunassa olevaan tietokantaan on tallennettuna testeihin liittyvä data. Simulointimallit, sekvenssit, järjestelmien väliset kommunikointitiedot yms. on tallennettuna kaikki erikseen. Testiautomaatio-ohjelmistossa luodaan testitapaukset, joissa viitataan tietokannassa oleviin tietoihin. Näin ei luoda ylimääräisiä kopioita samoista tiedoista. Samalla ylläpidettävyys helpottuu, kun tietokannassa oleviin tietoihin tehdyt muutokset päivittyvät automaattisesti kaikkiin testeihin.

Testitoiminnon erottamisella sen teknisestä toteutuksesta tarkoitetaan, että ihmisen ei tarvitse tietään miten varsinainen toiminto suoritetaan, kunhan se toimii halutusti. Esimerkiksi toimintona voi olla työkoneen puomin liikuttaminen tiettyyn asentoon, jolloin ihminen ei välitä mitä käskyjä testattavalle järjestelmälle tarvitsee lähettää, kunhan lopputuloksena puomi on liikkunut haluttuun asentoon. Käytännössä tämän toteutus riippuu toimintojen ja sekvenssien kuvaamiseen käytetystä ohjelmointikielestä, mutta yleisin tapa on käyttää makroja ja funktioita. Ne voidaan ohjelmoida toteuttamaan yksinkertaisia toimintoja, kuten puomin liikuttamista. Tällöin funktiolle tai makrolle annetaan parametrina arvo, johon puomin halutaan liikkuvan. Ylläpidettävyys helpottuu, kun järjestelmän muuttuessa muutoksia tarvitsee tehdä vain alemman tason funktioihin ja makroihin, jolloin varsinaiset testisekvenssit pysyvät muuttumattomina.

Automaattisella testausprosessin toteutuksella tarkoitetaan testiin liittyviä ohjelmia ja laitteistoa. Jotta laitteisto saadaan eriytettyä testiautomaatio ohjelmistosta, tarvitaan standardoitu rajapinta laitteiston ja ohjelmiston väliin. ASAM HIL API-standardi on heinäkuussa 2009 julkaistu standardi, joka pyrkii standardisoimaan tämän rajapinnan. Standardin uutuudesta johtuen, sitä käyttäviä laitteistoja tai ohjelmistoja ei kuitenkaan ole vielä saatavilla. [10]

Ohjelmistojen eriytettävyydellä tarkoitetaan taas sitä, että testausprosessin aikana käytettävät ohjelmat toimivat erillään testausprosessista. Esimerkiksi analysointi ohjelma ja sekvenssien suorittamisesta vastaava ohjelma on erillään testiautomaatio

ohjelmasta. Tämä on tärkeää, koska kehitys- ja testausprosessin eri vaiheissa pitää analysoida erilaisia signaaleja ja kommunikoida testattavaan järjestelmään eri tavoilla.

2.4. Testitapausten luominen

Testitapausten automaattinen generointi on erittäin tärkeä osa automaattista testausta. Optimitapauksessa testattaisiin kaikki mahdolliset sisääntulojen arvot erilaisina kombinaatioina. Tämä tarkoittaisi kuitenkin ääretöntä määrää testitapauksia, joiden testaaminen veisi äärettömän kauan. Kirjassa *The Art of Software Testing* [11] on kuvattuna testitapausten määrän kasvua. Siinä on esitettynä ohjelma, jossa on viisi loogista lauseketta. Näistä viidestä lausekkeesta saadaan 10^{14} erilaista suorituspolkua. Kun vielä otetaan huomioon että loogisten lauseiden määrä on oikeissa ohjelmissa huomattavasti isompi, niin kaikkien mahdollisien kombinaatioiden testaus ei ole mahdollista. Realistisempi vaihtoehto on pyrkiä valitsemaan testitapausten joukosta ne tapaukset, joilla pystytään suurimmalla todennäköisyydellä paljastamaan virheet.

Seuraavassa esitellään menetelmiä joilla pyritään luomaan testitapauksia järjestelmällisesti. Nämä tekniikat on jaettu funktionaaliseen testaukseen, kattavuustestaukseen ja grey box-testaukseen.

2.4.1. Funktionaalinen testaus

Funktionaalista testausta sanotaan usein myös black-box-testaukseksi [12]. Tällöin ei välitetä ohjelman sisäisestä rakenteesta tai toteutuksesta, vaan siinä kuvataan järjestelmää mustana laatikkona, jonka sisälle ei näe. Tällöin ainoa tapa testata järjestelmän toimivuutta on kokeilla tuottaako järjestelmä tietyillä sisääntuloilla määrityksien mukaiset ulostulot. Jotta voidaan olla täysin varmoja ohjelman toimivuudesta, tulisi kaikki sisäänmenot testata kaikilla mahdollisilla arvoilla sekä erilaisilla sisäänmenojen kombinaatioilla. Käytännössä tämä on mahdoton toteuttaa, sillä testitapauksia tulisi ääretön määrä. [11]

Esimerkkejä funktionaalisesta testauksesta ovat mm. stressitestaus, raja-arvo analyysi, suorituskykytestaus ja error guessing. Stressitestaus ja suorituskykytestaus keskittyvät molemmat arvioimaan järjestelmän toimivuutta, kun sitä kuormitetaan yli normaalien toimintarajojen. Tällöin voidaan esimerkiksi käyttää suurempia sisääntulojen arvoja kuin järjestelmän normaalikäytössä tulisi vastaan. Raja-arvo analyysissä taas testataan miten ohjelma toimii tiettyjen sisääntulojen raja-arvojen läheisyydessä. Esimerkiksi tiedettäessä sisääntulon mahdollinen maksimiarvo voidaan testata järjestelmän toimintaa tuon raja-arvon molemmilla puolilla. Error guessing tarkoittaa sananmukaisesti valistunutta arvausta missä kohtaa järjestelmää voisi olla virhe. Tämä perustuu pitkälti testaajan omaan kokemukseen.

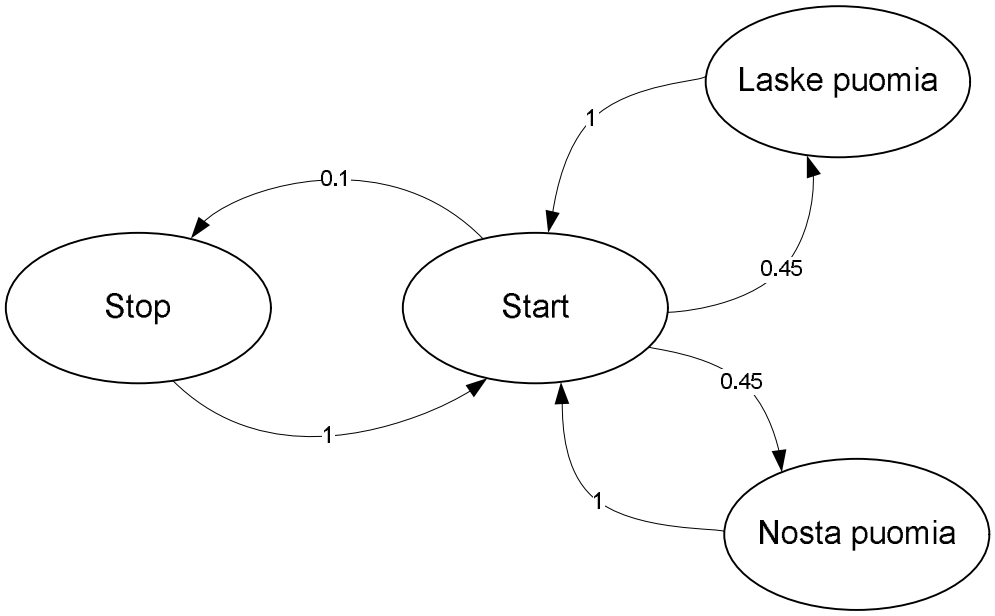
Kaikille funktionaalisen testauksen menetelmille on yhteistä se, että niiden kehitykseen ja analysointiin vaikuttaa ainoastaan järjestelmän I/O-rajapinta sekä järjestelmälle asetetut vaatimukset. Koska järjestelmän sisäistä toteutusta ei tarvitse

tuntea, voidaan funktionaalista testausta ruveta kehittämään samanaikaisesti itse ohjelman kanssa. [12]

Funktionaalisten testien luonnin automatisoinnissa voidaan käyttää joko tilastollista käyttötapausgenerointia tai evoluutiomenetelmää. Molemmat tavat pohjaavat jo olemassa oleviin pieniin käyttötapauspohjiin, joissa kuvataan lyhyt käyttötapausalkio. Tämä alkio voi esimerkiksi olla puomin liikuttaminen, napin painaminen tai moottorin käynnistäminen. Näitä alkiota yhdistämällä saadaan pitempiä käyttötapauksia, joita yhdistelemällä saadaan pitempiä testejä. Lisäksi alkion parametreja muuttamalla saadaan eri variaatioita samasta alkiosta, kuten puomin liikuttaminen eri nopeuksilla ja eri kohtiin.

Tilastollisen menetelmän avulla pystytään luomaan testitapauksia, jotka mallintavat oikeita käyttötapauksia. Se perustuu järjestelmän käyttöprofiiliin, jossa kuvataan miten järjestelmää käytetään ja kuinka usein tietyt käyttötapaukset esiintyvät käytössä. [9,13]

Tilastollista menetelmää voidaan käyttää tilamuotoisiin järjestelmiin. Niissä jokainen tila mallinnetaan käyttötapausalkioksi ja niihin liittyviin siirtymiin annetaan todennäköisyydet. Näiden todennäköisyyksien avulla voidaan generoida testitapauksia, jotka vastaavat mahdollisimman hyvin koneen oikeaa käyttöä. Kuvassa 8 on esitetty puomin tilakone, jonka perusteella on mahdollista luoda oikeaa vastaava työkierto.



Kuva 8. Puomin tilakonemalli testitapausten tilastolliseen generointiin.

Kuvan tapauksessa puomin ohjaus käynnistetään aina kun se ei ole käynnissä. Käynnissä olevaa puomin ohjausta ohjataan joko alas tai ylöspäin yhtä suurilla todennäköisyyksillä. Yhdessä kymmenestä tapauksesta ohjaus sammutetaan.

Toinen mahdollinen tapa on evoluutiomenetelmä. Siinä pyritään luomaan mahdollisimman hyviä testitapauksia, jotka löytävät mahdollisimman hyvin eri virheet. Evoluutioon perustuva menetelmä käyttää myös pieniä käyttötapauskuvauksia pohjanaan, joita sopivasti yhdistämällä saadaan toimiva testitapaus. Evoluutiotestauksessa vaaditaan hyvyysfunktio, jonka perusteella arvioidaan

testitapauksen paremmuutta. Se laskee testituloksista kuinka hyvin testitapaus paljastaa virheet ja tämän tiedon perusteella pyritään luomaan uusia parempia testitapauksia. Tämän hyvyysfunktion luominen voi olla hyvinkin vaikeaa, jonka takia menetelmä ei ole kovinkaan helppo käyttää.

2.4.2. Kattavuustestaus

Kattavuustestauksesta käytetään myös nimitystä white box-testaus [12]. Tällöin tunnetaan järjestelmän sisäinen rakenne ja toteutus, joita myös käytetään hyväksi testauksessa. Siinä tutkitaan järjestelmän loogista rakennetta ja mahdollisia suorituspolkuja, joita ohjelmassa tulee vastaan. Näiden avulla voidaan luoda testitapauksia, joissa kaikki loogiset operaatiot ja kaikki mahdolliset polut tulee suoritettua läpi ainakin kerran. Toteutuksen hyvänä puolena on, että tällöin testitapauksia tulee rajoitettu määrä, joten niiden testaaminen on ainakin teoriassa mahdollista. [11]

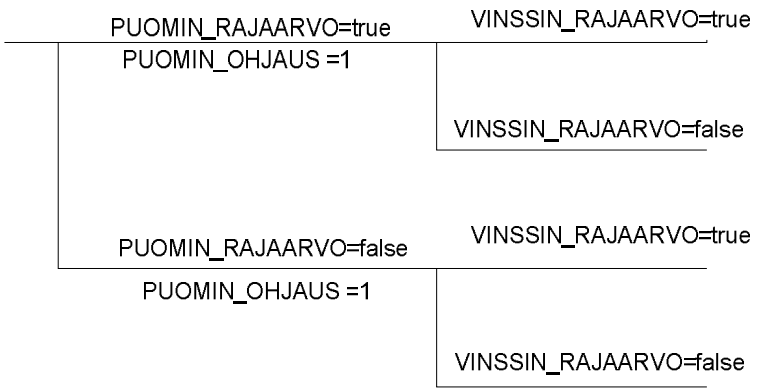
Kattavuustestauksessa käytettyjä testausmetodeja ovat esimerkiksi statement coverage(lausekattavuus), decision and branch coverage(haara- ja päätöskattavuus) sekä condition coverage(ehtokattavuus). Statement coverage tarkoittaa koodirivien kattavuutta. Se varmistaa, että jokainen koodirivi tulee käytyä läpi ainakin kerran.

Decision and branch coverage tarkoittaa haarojen kattavuutta. Käytännössä siinä tarkastetaan, että jokaisesta IF-lauseesta evaluoituu ainakin kerran TRUE ja FALSE arvot. Näin voidaan olla varmoja, että kaikki suoritushaarat käydään läpi.

Condition coverage tarkoittaa IF-lausekkeiden sisällä olevien lauseiden tarkastelua. Tällöin tarkastetaan, että kaikki mahdolliset IF-lausekkeen sisältämät TRUE ja FALSE yhdistelmät käydään läpi.

Testitapausten automaattinen generointi yllämainittuja menetelmiä käyttäen on mahdollista koodia analysoimalla. Jokainen suoritettava haara voidaan kuvata yhdellä testitapauksella ja jokainen IF lauseessa suoritettava looginen vertailu aiheuttaa uuden haaran. Kuvassa 9 on esitettynä ohjelman haarautuvuus, kun käytetään decicion and branch coverage-menetelmää.

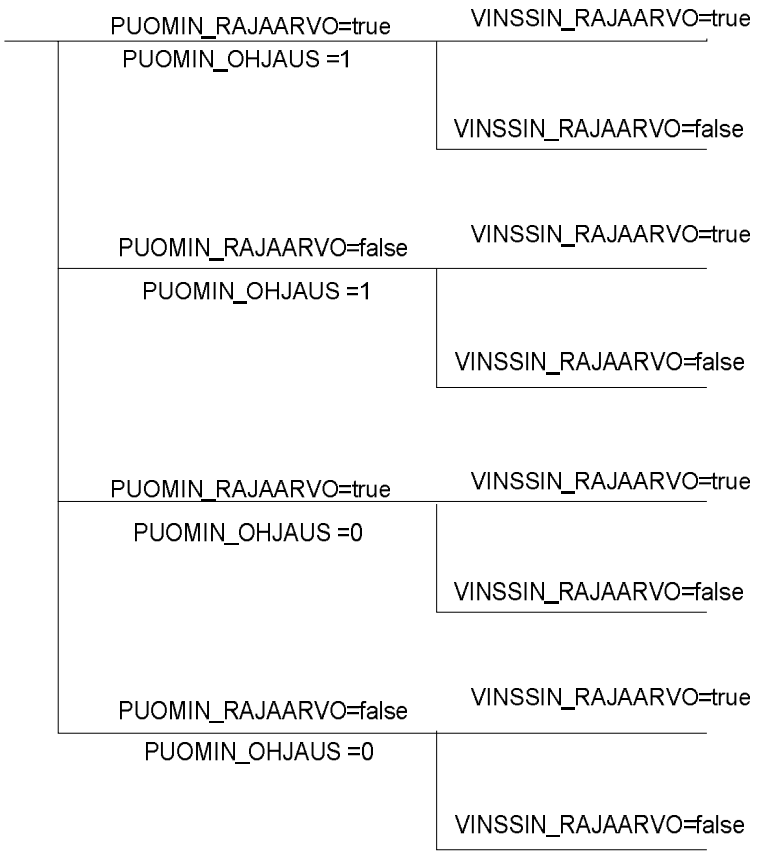
```
If PUOMIN_RAJAARVO && PUOMIN_OHJAUS>0
        PUOMIN_OHJAUS=0
If VINSSIN_RAJAARVO
        VINSSIN_OHJAUS=0
```



Kuva 9. Decision and branch coverage-menetelmän mukainen ohjelman haarautuvuus.

Ohjelmassa on kaksi IF-lausetta. Decision and branch coverage metodi ottaa huomioon vain näiden lauseiden totuusarvot, eli ei ole väliä miten if lauseesta evaluoituu TRUE ja FALSE arvot, kunhan näin tapahtuu. Menetelmän huonona puolena on, ettei kaikkia totuuslauseita testata. Kuvan 9 tapauksessa puomin ohjauksen vaikutusta ei testata ollenkaan. Jotta kaikki IF-lauseiden sisäisetkin ehtolauseet saadaan testattua, tulee käyttää condition coverage metodia, joka on esitettynä kuvassa 10.

```
If PUOMIN_RAJAARVO && PUOMIN_OHJAUS>0
        PUOMIN_OHJAUS=0
If VINSSIN_RAJAARVO
        VINSSIN_OHJAUS=0
```



Kuva 10. Condition coverage-menetelmän mukainen ohjelman haarautuvuus.

Nyt kaikki ohjelman sisältämät ehtolauseet testataan. Huonona puolena menetelmässä on testitapausten määrän kasvu.

Kattavuustestauksen menetelmien käyttäminen ohjausjärjestelmien HIL-testaamiseen vaatii, että otetaan huomioon signaalit jotka riippuvat ajasta ja muista signaaleista. Ohjelmistotestauksessa ei tarvitse huomioida että joidenkin parametrien muuttamiseen kuluu tietty aika ja että niitä pitää muuttaa käyttäen jotain toista signaalia apuna. Esimerkkitapauksissa olevan PUOMIN_RAJAARVO muuttujan muuttaminen vaatisi esimerkiksi puomin ohjauksen muuttamista niin että puomi saadaan ajettua raja-arvoon. Tällaiseen toiminnallisuuteen vaaditaan että mallin riippuvuudet on määritelty, jolloin tiedetään miten sisääntulot vaikuttavat ulostuloihin. Tämä puolestaan vaikuttaa signaalien generointiin niin, että signaalit generoivan järjestelmän tulee pystyä ohjaamaan signaaleja, käyttäen hyväksi näitä riippuvuuksia. Koodissa, josta testitapaukset generoidaan, on tietysti nämä riippuvuudet esitettynä, mutta ne tulee kertoa signaaligeneraattorin ymmärtämässä muodossa, joka on tapauskohtainen. Esimerkiksi GIMsim Sequence Generator-ohjelma käyttää alustustiedostoa, jossa nämä riippuvuuden on esitetty.

2.4.3. Grey Box testaus

Jos ohjausjärjestelmän sisäistä toteutusta ei täysin tiedetä, mutta sen sisäisiin muuttujiin päästään kuitenkin käsiksi, on kyseessä grey box-testaus. Se on yhdistelmä black- ja white box-testauksesta. Tällöin testitapauksien luonti suoritetaan järjestelmän määritelmistä, mutta testejä suoritettaessa ja analysoitaessa voidaan tutkia ulostulosignaalien lisäksi sisäisiä muuttujia.

Jotta sisäisiä muuttujia olisi mahdollista tutkia, tulee ohjausjärjestelmässä olla rajapinta, jonka kautta sen sisäisiin muuttujiin päästään käsiksi. Tämä vaatii tukea aina ohjausyksikön tasolta, jonka takia grey box-testauksen toteuttaminen ei ole mahdollista ilman ohjausyksiköiden valmistajien tukea. ASAM XCP ja ISO 15765 standardit tarjoavat mahdollisuuden päästä käsiksi ECU:n, Electronic Control Unit, muistissa oleviin tietoihin, mutta molemmat näistä on tarkoitettu lähinnä autoteollisuudessa käytössä oleviin ohjausyksiköihin. Kyseisiä standardeja ei myöskään ole suoraan tarkoitettu grey box-testaukseen, jonka takia niiden tarjoamiin mahdollisuuksiin tulisi perehtyä tarkemmin ennen kuin voidaan todeta niiden soveltuvuus kyseiseen menetelmään. Tämän työn puitteissa se ei kuitenkaan ollut mahdollista.

2.5. Signaalien generointi

Testien suorittamiseksi tulee simulointimalleja kontrolloida ja niihin tulee olla mahdollista lähettää signaaleja. Tekniikka, jota kulloinkin käytetään, riippuu paljolti käytettävästä laitteistosta. Signaalien generointiin on löydettävissä kolme toisistaan eroavaa tapaa.

- Signaalien lukeminen tiedostosta simulaattorilta
- Signaalien saaminen simulaattorilla toimivasta signaaligeneraattorista
- Signaalien saaminen ulkomaailmassa toimivasta signaaligeneraattorista

Signaalien lukeminen suoraan tiedostosta on erittäin tarkka tapa toistaa samoja signaaleja eri testikerroilla. Tämä tapa soveltuu esimerkiksi tuotantotestaukseen, jolloin ajetaan samoja testejä ohjausjärjestelmän eri tuotantokappaleille ja vertailuun käytettävät signaalit ovat myös valmiina. Menetelmän huonona puolena on tiedostojen luominen. Ne voidaan luoda joko tyhjästä tai tallentaa jollain muulla tavalla suoritetun testin ohjaukset. Menetelmän ideana on, että jokaisella simulointiaskeleella luetaan tiedostosta kyseistä askeletta vastaava signaalin arvo. Esimerkkinä tästä tavasta voidaan mainita xPC Target-järjestelmän FromFile-lohko, jonka avulla voidaan lukea tiedostossa olevaa dataa.

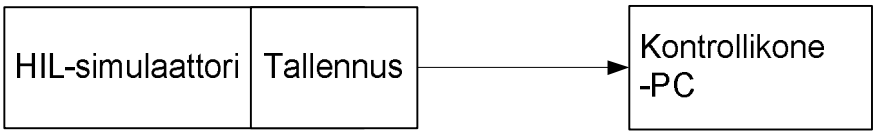
Signaaleja voidaan myös luoda signaaligeneraattoreilla, jotka toimivat joko simulointikoneen yhteydessä tai simuloinnin ulkopuolella. Kun signaaligeneraattori toimii simulaattorin kanssa samalla koneella, pystytään sekvenssejä kontrolloimaan tarkemmin, sillä tällöin sekvenssi toimii samalla askelvasteella kuin simulointimalli eikä datan kuljetusviiveistä tarvitse huolehtia. Signaalien generointi voidaan tällöinkin jakaa vielä kahteen alaluokkaa, jolloin signaaligeneraattori on sisällytettynä malliin tai toimii

reaaliaikakoneella erillisenä osana. Jälkimmäinen tapa vaatii tukea reaaliaikakoneelta, kun taas ensimmäinen tapa voidaan toteuttaa lähes missä ympäristössä vain. DSpace reaaliaikasimulaattorilla signaalien generointi on toteutettu stimulus tekniikalla. Siinä reaaliaikakoneessa toimii signaaligeneraattori, joka luo signaaleja käyttäjän antaman python scriptin mukaisesti.

Signaaligeneraattori voi myös sijaita simulaattorin ulkopuolella, jolloin simulaattorin tekniikka ja toteutus eivät rajoita sen toimintaa yhtä paljoa. Tällöin joudutaan kuitenkin pois reaaliaikaympäristöstä ja kommunikaatio simulaattorien kanssa tapahtuu useimmiten lähiverkon välityksellä, jotka molemmat aiheuttavat viiveitä sekvenssien ajamiseen. Tämä tapa tuo mukanaan kuitenkin joustavuutta simulaattorin ohjaamiseen, jolloin ohjausjärjestelmää voidaan ohjata ihmismäisesti. Tämä ominaisuus on tarpeellinen erityisesti käyttötapaustestauksessa.

2.6. Datan keräys

Datan keräys-menetelmiä on myös useita, mutta tästäkin voidaan eritellä kolme päätapaa. Niiden käyttö riippuu käytössä olevasta laitteistosta ja keräykselle asetetuista vaatimuksista. Ensimmäinen menetelmä on datan tallennus reaaliaikakoneelle. Sen periaate on esitetty kuvassa 11.



Kuva 11. Datan reaaliaikainen tallennus reaaliaikakoneelle.

Datan tallennus reaaliaikakoneelle vaatii, että reaaliaikakäyttöjärjestelmä tukee toimintoa. Data voidaan tallentaa joko tietokoneen muistiin tai kovalevylle, josta ne voidaan myöhemmin hakea kontrollikoneelle analysoitaviksi. Tämän tavan hyvänä puolena on, että se ei aiheuta ylimääräistä liikennettä verkkoon ja häiritse näin mallien välistä kommunikointia. Huonona puolena taas on sen rajoittaneisuus, koska tarkempia määrityksiä sille, mitä ja milloin tallennetaan, ei voi asettaa tai se on hyvin työlästä.

Kun reaaliaikakone ja kontrollikone pystyvät kommunikoimaan keskenään, voidaan tallennettavat signaalit välittää kontrollikoneelle simuloinnin aikana. Tämä aiheuttaa kuitenkin liikennettä verkkoon, joka saattaa haitata mallien välistä kommunikointia. Jos kontrollikoneella ja reaaliaikakoneella käytetään saman valmistajan ohjelmia, on mahdollista, että ne pystyvät kommunikoimaan keskenään helposti, jolloin tallennettavat tiedot voidaan valita suoraan kontrollikoneella toimivasta ohjelmasta. Esimerkiksi Matlab pystyy kommunikoimaan ja välittämään tieto tällä tavalla ja sen periaate on esitetty kuvassa 12.

Kuva 12. Datan välitys ja tallennus kontrollikoneelle reaaliaikakäyttöjärjestelmän toimesta.

Jos taas reaaliaikakone ja kontrollikone eivät pysty kommunikoimaan yllä mainitulla tavalla, voidaan verkon kautta tulevat signaalit tallentaa vielä erikseen. Tämä vaatii, että mallit lähettävät halutut signaalit esimerkiksi UDP paketteina, jotka vastaanotetaan kontrollikoneella. Menetelmän periaate on esitetty alla olevassa kuvassa 13.

Kuva 13. Datan välitys ja tallennus kontrollikoneelle verkon kautta.

Huonona puolena on se, että signaalien lähetys pitää lisätä suoraan malliin, jonka jälkeen mallit pitää kääntää uudestaan, tämä taas lisää usein käsin tehtävää työtä, vaikka mallien kääntäminen olisikin mahdollista automatisoida. Hyvänä puolena taas on, että tällöin tiedetään tarkalleen mitä liikennettä verkossa kulkee, jolloin esimerkiksi datan tallennustarkkuutta voidaan muuttaa verkkoresurssien vapauttamiseksi. Kuvan 12 tapauksessa datan välityksen yksityiskohdat eivät ole tiedossa, joten tarkkaa kuvaa datan välityksen vaatimista resursseista on mahdotonta saada.

2.7. Analysointi

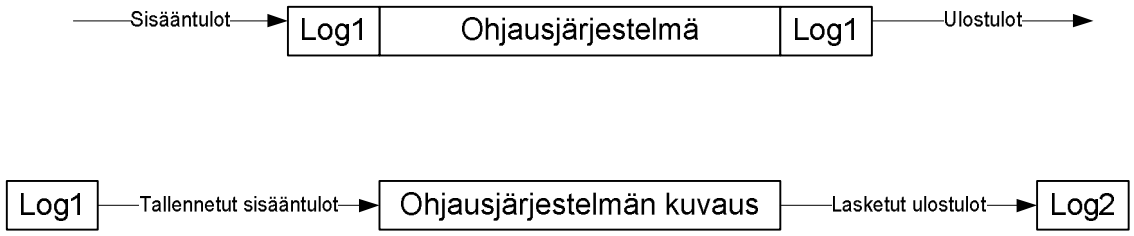
Kun testit on ajettu ja haluttu data on kerätty talteen, pitää data analysoida. Analysointiin on kehitetty erilaisia menetelmiä ja algoritmeja ja oikean analysointitavan käyttäminen onkin erittäin tärkeää. Väärän metodin valitseminen voi johtaa siihen, ettei ohjausjärjestelmässä olevaa vikaa löydetä. Testissä käytettävä metodi tai metodit tulisi lisäksi valita jo testin luontivaiheessa, jotta ne voidaan suorittaa testin suorituksen päätyttyä automaattisesti.

Testien tulosten automaattisen analysoinnin aikaansaamiseksi on helpointa käyttää vertailevaa analysointia. Tällöin kahta signaalia verrataan keskenään ja todetaan ovatko ne keskenään samanlaiset. Tietokoneella tehty kahden signaalien vertailu on helppo toteuttaa, koska niiden vastaavuudet on mahdollista osoittaa toleranssien ja raja-arvojen vertailujen avulla. Signaalien vertailua suoritetaan erityisesti regressio- ja back-to-back-testauksissa, joissa vertailuun käytettävät signaalit on mahdollista saada suoraan aikaisemmista testeistä.

Jos valmiita vertailuun tarkoitettuja signaaleja ei ole olemassa, pitää ne luoda jollain tavalla. Tällöin tulee kysymykseen ohjausjärjestelmän kuvauksen käyttö. Ohjausjärjestelmän kuvaus on ohjausjärjestelmän määrityksien perusteella tehty

ohjelmointikielinen kuvaus sen toiminnasta. Kuvassa 14 on esitetty ohjausjärjestelmän kuvauksen käytön periaate.

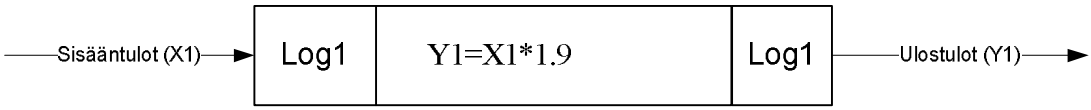
Testauksen aikana tallennetaan ohjausjärjestelmälle tulevat ja sieltä lähtevät signaalit(Log1). Ohjausjärjestelmälle sisääntulevia signaaleja käytetään sitten sisääntuloina ohjausjärjestelmän kuvaukselle, josta saadaan vertailusignaalit(Log2).



Kuva 14. Ohjausjärjestelmän kuvauksen käyttö vertailusignaalien generointiin.

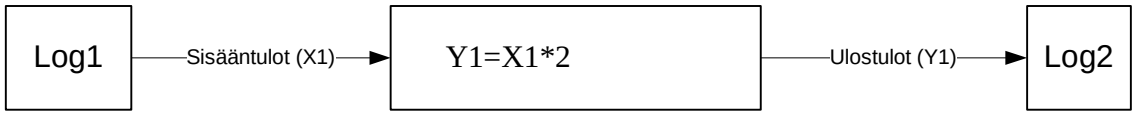
Ohjausjärjestelmän kuvaus on kirjoitettu ohjelmointikielellä kuten Python tai Matlab. Sen ajamiseen tarvitaan oma ohjelmansa, joka hakee lokitiedostosta(Log1) tarvittavat signaalit ja ajaa ohjausjärjestelmän kuvauksen kyseisillä sisääntuloilla. Tämän jälkeen ulostulot kirjoitetaan omaan lokitiedostoon(Log2), jota voidaan vertailla alkuperäisiä tuloksia vastaan(Log1).

Esimerkkitapauksessa voisi olla kuvassa 15 esitetty tilanne, jossa ohjausjärjestelmän sisääntulona on muuttuja X1 ja ulostulona muuttuja Y1. Testattava ohjausjärjestelmä kertoo sisääntulon kertoimella 1.9, josta saadaan ulostulo.



Kuva 15. Testattavan ohjausjärjestelmän periaatekuva esimerkkitapauksessa.

Ohjausjärjestelmän tulisi määritysten mukaan kuitenkin toimia kertoimella 2, jolloin kyseessä olisi kuvan 16 kaltainen tilanne. Kaava Y1=X1*2 toimii siis ohjausjärjestelmän kuvauksena.



Kuva 16. Ohjausjärjestelmän kuvauksen periaatekuva esimerkkitapauksessa.

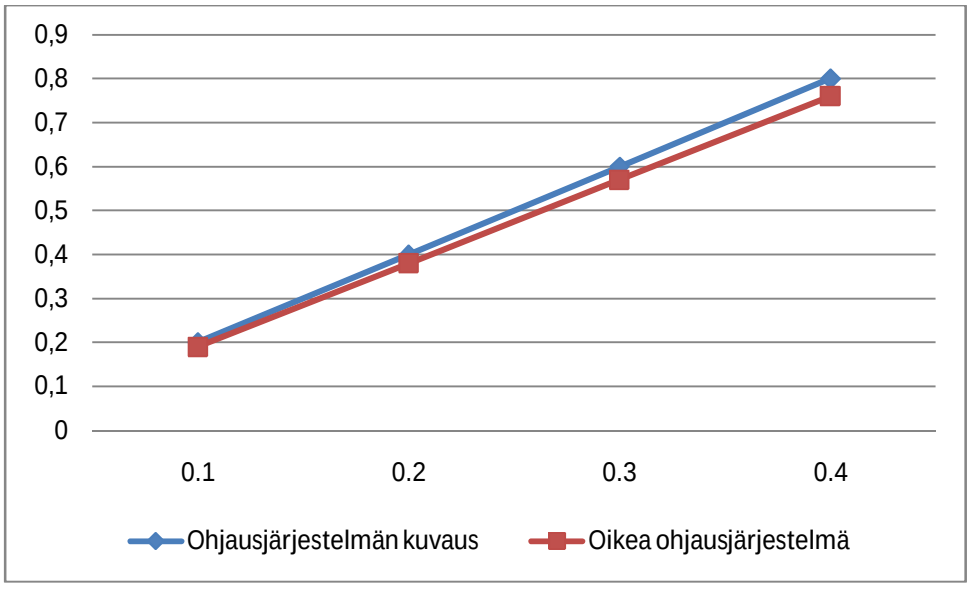
Sisääntulona X1 voidaan käyttää ramppisignaalia, jossa arvot kasvavat lineaarisesti. Alla on esitetty kuvan 15 ohjausjärjestelmällä saatu lokitedosto, jossa sisään- ja ulostulon arvot on esitetty sarakkeittain ja signaalit on tallennettu 0.1 sekunnin välein.

Time	X1	Y1
0.0000	0.0000	0.0000
0.1	0.1	0.19
0.2	0.2	0.38
0.3	0.3	0.57
0.4	0.4	0.76

Ohjausjärjestelmän kuvauksen ajamiseen käytetään samoja sisääntuloja. Jokainen sisääntulo lasketaan ohjausjärjestelmän kuvauksen mukaisesti ja ulostulot tallennetaan uuteen lokitiedostoon, jolloin saadaan alla olevan kaltainen lokitiedosto.

Time	X1	Y1
0.0000	0.0000	0.0000
0.1	0.1	0.2
0.2	0.2	0.4
0.3	0.3	0.6
0.4	0.4	0.8

Näitä kahta tiedostoa voidaan sitten verrata keskenään, kuten on tehty kuvassa 17. Siinä voidaan huomata miten signaalit lähtevät eroamaan toisistaan. Näitä kahta signaalia on helppo verrata keskenään automaattisesti, kun ottaa kahdesta lokitiedostosta samaa aikahetkeä vastaavat arvot. Tällöin kuvassa näkyvä virhe saadaan havaittua, kun virheen toleranssiksi asetetaan alle 10 prosentin eroavaisuus.

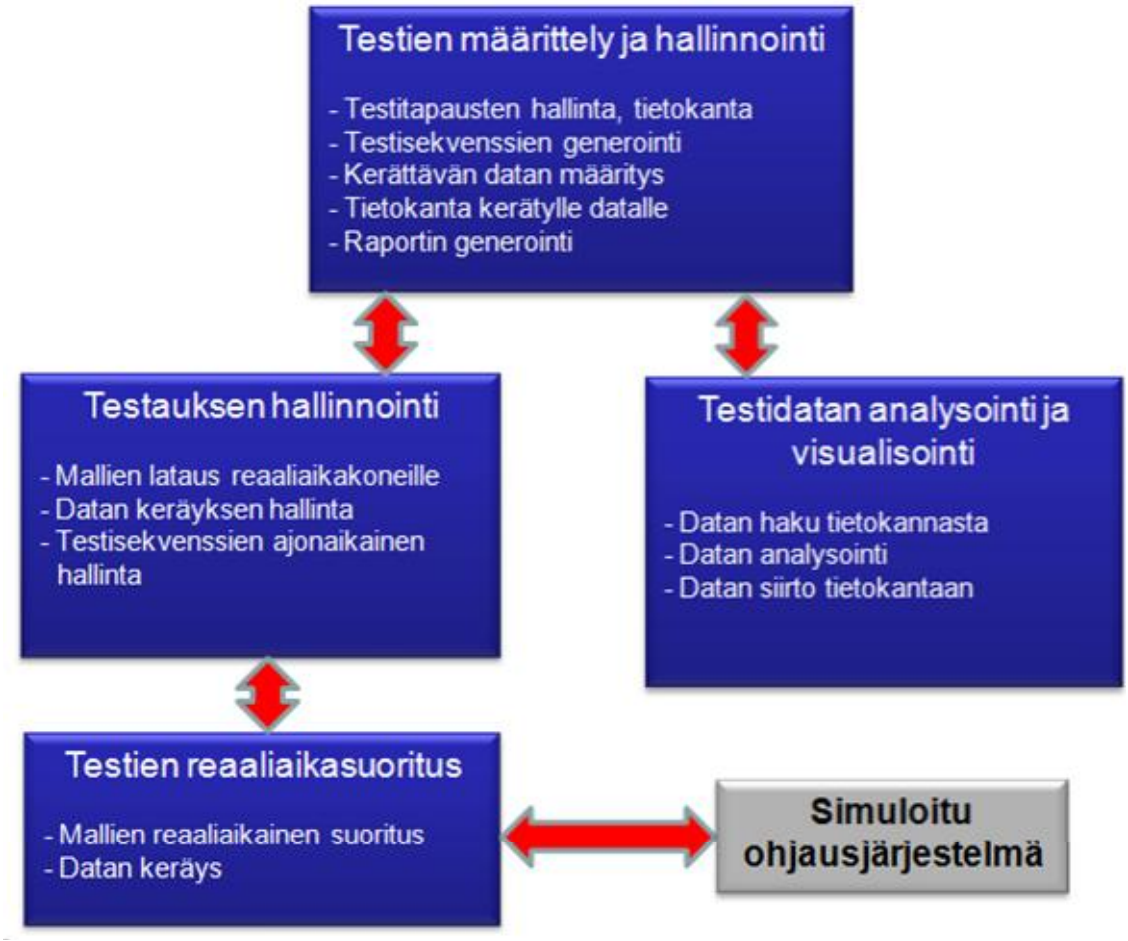


Kuva 17. Signaalien vertailu.

Samanlaista tekniikkaan on käytetty IHA:n automaattisen testauksen konseptin toteutuksessa. Sen toimintaa on kuvattu tarkemmin luvussa 5, case Avantin yhteydessä.

3. AUTOMAATTISEN TESTAUKSEN KONSEPTI

Tässä kappaleessa kuvataan automaattisen testauksen konsepti, joka kehitettiin ohjausjärjestelmätestaukseen (kuva 18). Konseptissa kuvataan automaattisessa testauksessa tarvitut perusvaiheet. Vaiheet on pyritty kuvaamaan mahdollisimman abstraktilla tasolla, jossa ei oteta kantaa konseptin toteutukseen. Näin mahdollistetaan erillisten osien toteuttaminen eri ohjelmistoilla. Luvussa 3.1 on kuvattuna konseptin toteutus IHA:ssa. Se on tehty omilla järjestelmillä käyttäen hyväksi sekä Matlab-ohjelmistoa ja mahdollisia avoimen lähdekoodin ohjelmistoja.



Kuva 18. Automaattisen testauksen konsepti. [3]

Ohjausjärjestelmien testaus on konseptissa jaettu neljään vaiheeseen. Ensimmäisenä on testien hallinnointi. Tässä vaiheessa määritellään mitä testataan, miten testataan ja miksi testataan. Se pitää sisällään kaikkien testien määritelmät ja kaikki tarvittavat

tiedot, joita testauksen suorittamiseksi tarvitaan. Myös testeistä saatu data ja analysoinnin tulokset tallennetaan aikanaan osaksi testiä.

Testitapaukset tulee olla jaettuna loogisiin kokonaisuuksiin, jolloin testien hallinta helpottuu. Esimerkiksi tiettyä ohjausjärjestelmän osaa testaavat testit on asetettu saman kokonaisuuden alle. Testitapauksien ja siihen liittyvien tietojen tallentamiseen on hyvä käyttää tietokantaa sekä ohjelmistoa, jolla tätä tietokantaa pystyy muokkaamaan graafisen käyttöliittymän kautta. Tietokannan käyttäminen myös helpottaa usean käyttäjän yhtäaikaista toimintaa testauksen parissa, koska tällöin tietoihin pääsee käsiksi tietoverkon välityksellä useasta paikasta yhtä aikaa.

Testitapausten tulee sisältää määrättyjä tietoja, jotta niiden suorittaminen automaattisesti olisi mahdollista. Simulaatiomallien ja niihin liittyvien määrittelytiedostojen tulee olla liitettynä osaksi testiä. Määrittelytiedoissa voidaan kertoa malleissa käytettävät parametrit, simuloinnissa käytettävä askelaika, datan keräyksessä käytettävä askelaika sekä simulointikoneiden yhteystiedot.

Testin aikana voidaan kerätä huomattava määrä dataa. Kerättävä data tulee olla määriteltynä testitapauksessa. Usein kerätään samojen signaalien dataa, jolloin on hyvä että signaalit on ryhmitelty loogisesti. Datan keräystekniikasta riippuen signaaleita voidaan ryhmittää joko nimeämällä tai ryhmittämällä signaaleja erillisiin lohkoihin simulointimallissa. Ylimääräisen datan tallentamista tulee myös välttää, jotta analysointivaihe helpottuu.

Kun testit on määritelty, voidaan siirtyä niiden suorittamiseen. Jotta testausjärjestelmällä voidaan suorittaa automaattista testausta, tulee sen pystyä ajamaan useita testejä peräkkäin. Tällöin ohjelmisto huolehtii oikeiden simulointimallien lataamisesta simulointikoneille testien yhteydessä. Ohjelmisto alustaa malliin liittyvät määrittelyt, kuten parametrien muutokset, simulointiajan, askelajan yms. Tämän lisäksi ohjelmisto vastaa testisekvenssien oikea-aikaisesta suorittamisesta sekä datan keruusta ja alustamisesta. Testeissä käytettävät mallit voivat toisinaan kaatua, joka voi pahimmillaan pysäyttää testaamisen kokonaan. Testienhallinta ohjelmiston tulee siis pysytä myös vastaamaan virhetilanteisiin, jotta testauksen aikana ei tarvita ihmisen valvontaa.

Kolmantena vaiheena konseptissa on testien reaaliaikainen kontrollointi. Ohjausjärjestelmiä tulee pystyä ohjaamaan älykkäästi, jotta voidaan tuottaa aidon kaltaisia testitapauksia. Tällöin tarvitaan korkean abstraktiotason ohjelmointikieli, jolla pystytään hallitsemaan ohjaimia ja suorittamaan testisekvenssejä. Ohjelmointikielen tulisi myös sisältää makroja ja funktioita. Niiden avulla pysytään tuottamaan älykkäitä kokonaisuuksia, joilla voidaan esim. ajaa ohjausjärjestelmä tiettyyn tilaan, yhdellä funktiokutsulla. Tämä helpottaa sekvenssien kirjoittamista huomattavasti. Makrot ja funktiot eivät myöskään ole sidottuja yhteen testitapaukseen vaan niitä voidaan käyttää useissa eri tapauksissa. Ohjelmointikielen tulee pystyä siis kommunikoimaan simulointimallien kanssa, joka taas mahdollistaa säätimien rakentamisen ohjelmointikieleen.

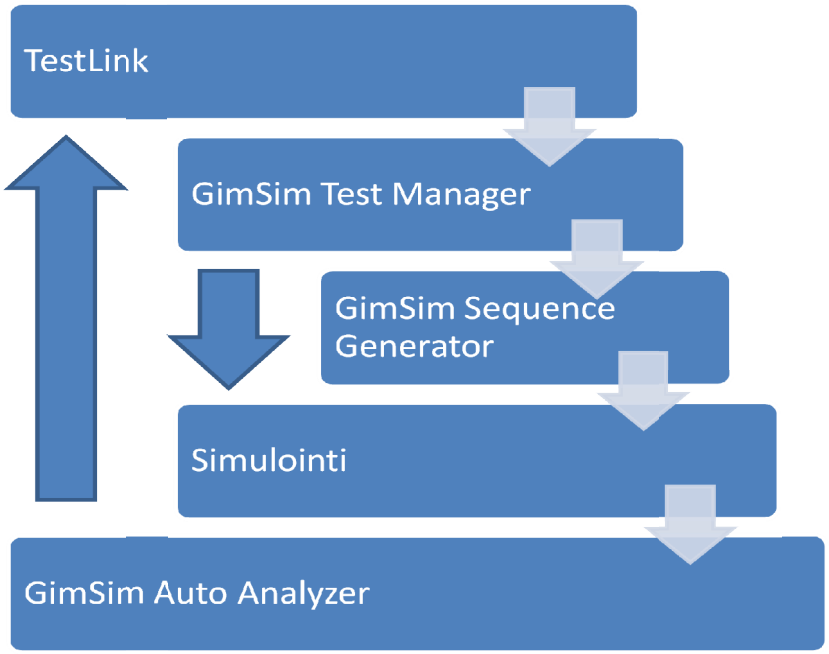
Konseptin neljäs vaihe on testien analysointi. Kun haluttu data on kerätty ja tallennettu, se analysoidaan erillisellä ohjelmistolla. Analysointiin on olemassa monia erilaisia signaalin prosessointi- ja vertailumetodeja. Käytettävä analysointimenetelmä riippuu pitkälti testitapauksesta ja tallennetuista signaaleista. Analysoinnissa saatu tieto tallennetaan taas takaisin testien hallinnoinnista vastaavaan tietokantaan. Näin sekä testitapaukset sekä niiden suoritustulokset löytyvät samasta paikkaa.

Analysoinnin jälkeen testausprosessi on valmis, mutta testauksen etenemistä ja tulosten tarkastelua varten tarvitaan raportointityökaluja. Nämä työkalut on toteutettu testien hallinnoinnista vastaavassa vaiheessa. Testaaja pystyy käyttämään näitä työkaluja jatkotestauksessa, kun tutkitaan löytyneitä virheitä.

3.1. Konseptin toteutus

IHA:ssa on toteutettu konseptin mukainen järjestelmä Matlab- ja Simulink-ympäristöön. Matlab ja Simulink valittiin, koska laitoksella oli valmiiksi kokemusta kyseisestä ympäristöstä ja sen tiedettiin olevan tarpeeksi joustava mahdollisten muutosten varalle. Lisäksi reaaliaikasimulointiin käytetty xPC Target-ympäristö voidaan toteuttaa normaaleilla PC- laitteilla, jolloin laitteistokustannukset pysyivät kurissa.

TINAT-projektin alussa tehdyssä HIL laite- ja palvelutarjonta raportissa huomattiin, että kaupallisilta toimijoilta ei löydy valmiita ohjelmistoja, joilla konsepti voitaisiin toteuttaa xPC Target-ympäristössä [5]. Tämän takia jouduttiin suuri osa ohjelmistoista kehittämään itse. Kuvassa 19 on esitettynä konseptin toteutuksessa käytetyt ohjelmistot ja niiden työsykli.



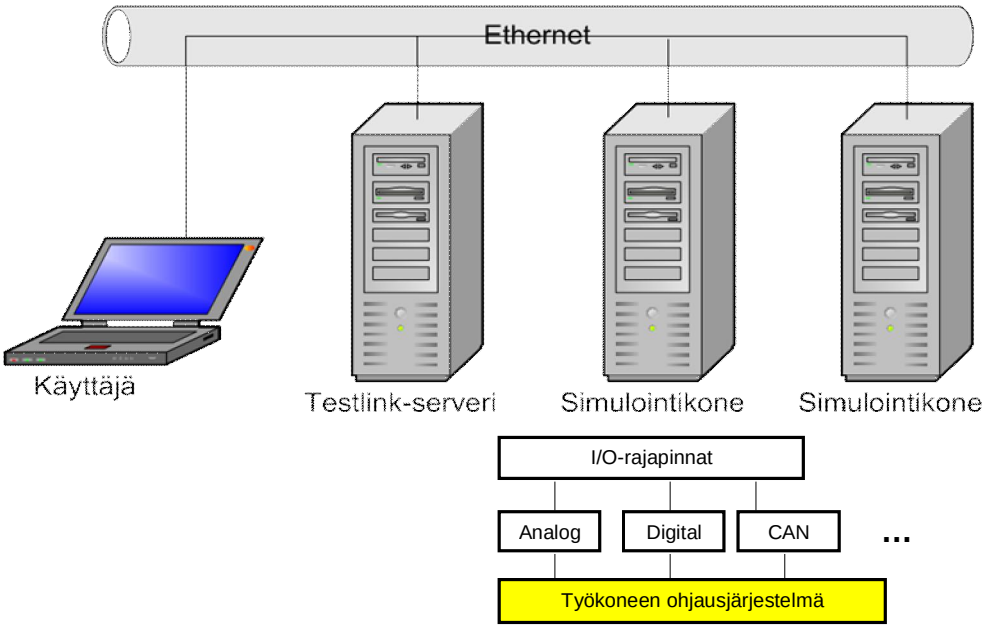
Kuva 19. Automaattisen testauksen konseptin toteutus IHA:ssa.

Konseptin toteuttamiseksi tehtiin tiivistä ryhmätyötä IHA:ssa, mutta kehittämisen selkiyttämiseksi annettiin eri henkilöille päävastuualueita. Tässä työssä keskitytään

TestLink-ohjelmiston soveltamiseen järjestelmään sekä GIMsim Test Manager-ohjelmaan. GIMsim Sequence Generator-ohjelmaa käsitellään Petri Mantereen diplomityössä ja GIMsim Auto Analyzer-ohjelmaa käsitellään Arvi Vallaksen diplomityössä.

3.1.1. Käytetty laitteisto

Reaaliaikasimulaatioympäristönä käytettiin MathWorksin xPC Target-järjestelmää. Laitteistona toimivat normaalit tietokoneet, joihin asennettiin xPC Target-käyttöjärjestelmä. Järjestelmän etuna on sen joustavuus ja laitteiston saatavuus. Useita koneita voi olla linkitettynä yhteen, jolloin raskaita simulointimalleja voidaan jakaa useammalle koneelle. Tällöin simulointikoneiden ei tarvitse olla yhtä tehokkaita, joka taas säästää laitteistokustannuksissa. IHA:ssa käytetyn laitteiston periaatekuva on esitettynä kuvassa 20.



Kuva 20. IHA:n testauslaitteiston periaatekuva.

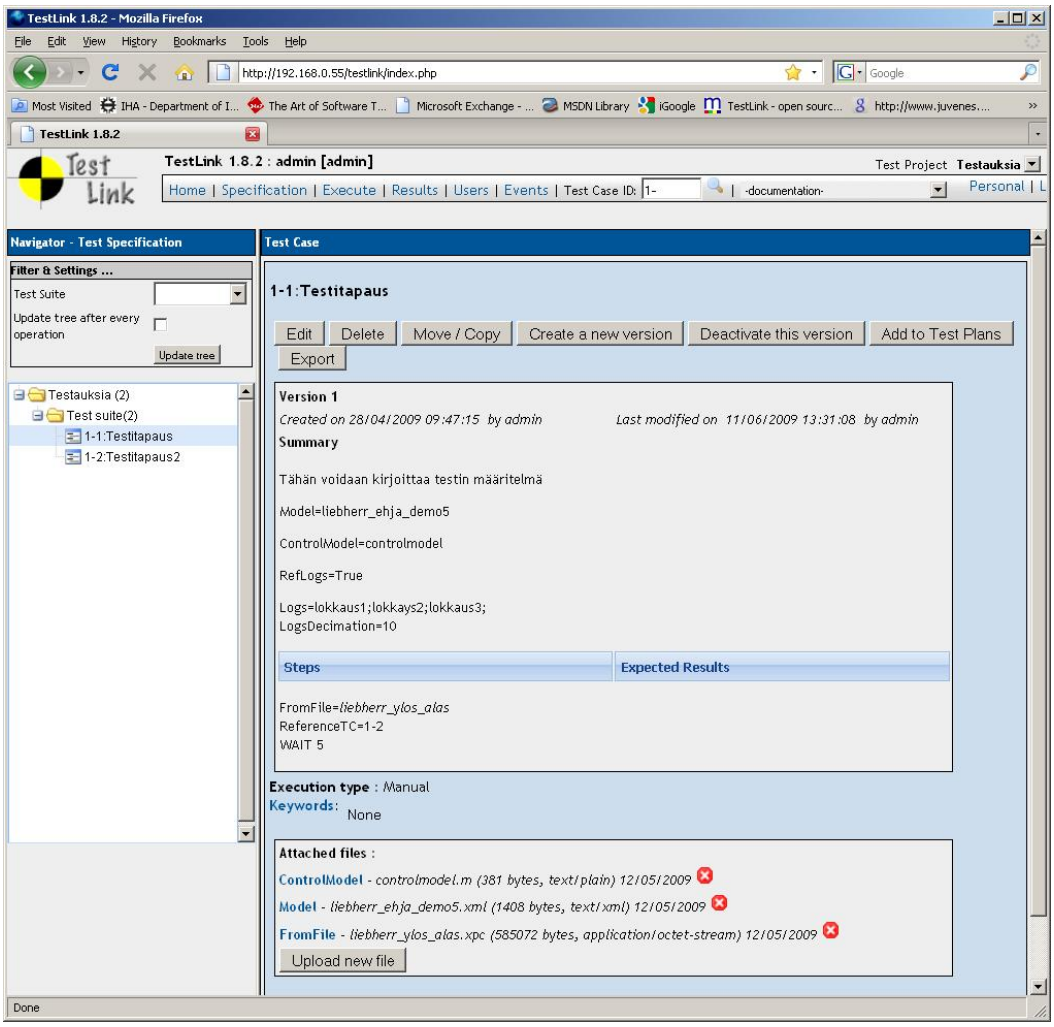
Simulointikoneita voi järjestelmässä olla siis useita. Yksi kone toimii rajapintana ohjausjärjestelmään ja sen simulointimallia kutsutaankin I/O- malliksi. I/O- koneessa voi olla ohjausjärjestelmästä riippuen analogi-, digitaali- tai vaikka CAN- liitäntöjä. Simulointikoneet ovat yhteydessä toisiinsa lähiverkon välityksellä. Eri koneilla toimivien mallien kommunikointi keskenään on toteutettu mallien sisäisellä toteutuksella. TestLink ohjelmistoa käytettiin testien hallinnointiin. TestLink on serveri pohjainen testien hallinta-ohjelmisto. Serveri, jossa ohjelmisto toimii, on kytkettynä samaan lähiverkkoon kuin simulointikoneetkin. Käyttäjän tietokone, jossa ovat testien ajamisesta ja analysoinnista vastaavat ohjelmat, liitetään myös samaan lähiverkkoon.

3.1.2. Testien määrittely

Testien määrittelyyn ja hallinnointiin valittiin TestLink ohjelmisto. Se on vapaan lähdekoodin ohjelmisto-projekti, jonka tarkoituksena on tarjota testien hallinnointi ohjelmisto. ”TestLink enables easily to create and manage Test cases as well as organize them into Test plans. These Test plans allow team members to execute Test cases and track test results dynamically, generate reports, trace software requirements, prioritize and assign tasks” [14]. Lisäksi ohjelmistossa on XML- RPC rajapinta, jonka kautta saadaan yhteys TestLinkiin. Tämä rajapinta mahdollistaa automaattisen testauksen, kun kaikki testit voidaan ladata ja niiden tulokset palauttaa lähiverkon välityksellä.

TestLinkissä testaus on jaettuna järjestelmän vaatimusten hallintaan, testien hallintaan ja testien suoritukseen. Vaatimusten hallinta tarkoittaa ohjausjärjestelmälle asetettujen vaatimusten kirjaamista järjestelmään ja testien liittämistä näihin vaatimuksiin. Jokaisen testitapauksen kohdalla voidaan siis määrittää mitä vaatimuksia siinä testataan.

TestLink ei aseta tarkkoja vaatimuksia testitapausten määrittämiseen. Se tarjoaa kolme tekstikenttää, johon voidaan kirjoittaa testitapauksen määrittely. Automaattisen testauksen konseptin tarpeisiin luotiin säännöt, joiden avulla näihin tekstikenttiin voidaan kirjata kaikki tarvittava tieto testin suorittamiseksi. Yhteen testitapaukseen tulee siis määrittää käytettävät simulointimallit, vertailussa käytettävä kontrollimalli, tallennettavat signaalit, signaalien tallennuksessa käytettävä askelaika, analysoinnissa käytettävät parametrit ja testisekvenssi. Testisekvenssi kirjoitetaan Steps-osioon, mutta muut tiedot kirjoitetaan Summary-osioon. Jokaiselle tiedolle on määrätty avainsana jonka perään voidaan haluttu tieto kirjoittaa. Esimerkiksi mallitiedoston nimi kirjoitetaan ”Model=”-avainsanan perään. Kuvassa 21 on esitettynä TestLink ohjelmiston testitapausten luontinäkymä.



Kuva 21. TestLink-ohjelman testitapausten luontinäkymä.

Testisekvenssi voidaan luoda kahdella tapaa. Ensimmäinen tapa on käyttää valmiina olevaa FromFile sekvenssitiedostoa. Se sisältää ohjaussignaalit, jotka on tallennettu Simulinkin FromFile-lohkon ymmärtämään muotoon. FromFile sekvenssitiedosto voidaan luoda aikaisemmin suoritetun testin ohjaussignaaleista, olettaen että kaikki tarvittavat signaalit on tallennettu. Toinen tapa on käyttää TINAT-projektissa kehitettyä ohjausjärjestelmien ohjaamiseen tarkoitettu korkean tason skriptikieltä.

Toistaiseksi testitapaukset kirjoitetaan käsin, mutta XML-RPC rajapinnan kautta on mahdollista myös tuoda järjestelmään testitapauksia. Tämä mahdollistaa testitapausten automaattisen generoinnin erillisellä ohjelmalla, jolloin ne voidaan tuoda TestLinkiin kyseisen rajapinnan kautta. Kun testitapauksia kirjoitetaan käsin, voidaan käyttää valmiita pohjia, joissa on täytettynä osa tiedoista jo valmiiksi. Myös valmiiden testitapausten kopioiminen on mahdollista.

Koska suurien tiedostojen siirtäminen XML-RPC rajapinnan kautta on hidasta, ei malleja ja lokitiedostoja tallenneta suoraan TestLinkin tietokantaan, vaan erilliseen jaettuun kansiorakenteeseen. Siellä eri kansioihin on jaoteltuna seuraavat tiedot:

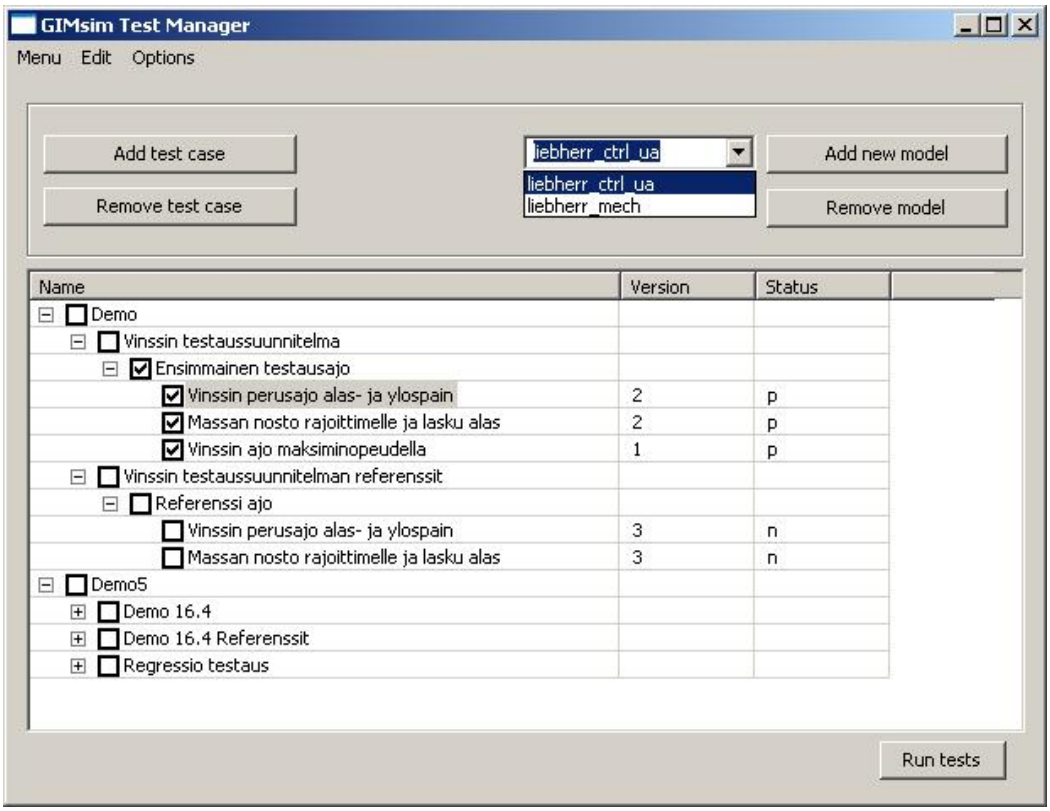
- Simulointimallit
- XML- koontitiedostot

- Sequence Generatorin .ini-tiedostot
- Sequence Generatorin makrot ja sekvenssit
- FromFile sekvenssit
- Testidata
- Control Model-vertailu mallit

Tietojen eriyttäminen toisistaan parantaa myös eri tiedostojen uudelleenkäytettävyyttä ja koko testausprosessin vikasietoisuutta. Jos simulointimallissa havaitaan virhe, voidaan ainoastaan kyseinen malli päivittää eikä muihin tiedostoihin tarvitse kajota.

3.1.3. Testien suoritus

Testien suorittamiseksi tulee testit ladata TestLinkistä erilliseen ohjelmaan, joka vastaa testauksen hallinnoinnista. Tätä tarkoitusta varten kehitettiin GIMsim Test Manager-ohjelman. Vastaisuudessa kyseisestä ohjelma käytetään nimitystä Test Manager. Se lataa testitapausten tiedot TestLinkistä, jonka jälkeen ne voidaan suorittaa. Kuvassa 22 on esitettynä Test Managerin pääikkuna. Siinä näkyy TestLinkistä ladatut testitapaukset järjestettynä projektien, testaussuunnitelmien ja kehitysversioiden alle. Ohjelma ja sen toiminta on esitetty tarkemmin luvussa 4.



Kuva 22. GIMsim Test Manager-ohjelman pääikkuna.

Suoritettavat testit voidaan valita listasta. Varsinaisesta testisekvenssien reaaliaikaisesta suorittamisesta vastaa GIMsim Sequence Generator-ohjelmisto, josta

käytetään vastaisuudessa nimitystä Sequence Generator. Se on tarkoitettu simulaattoriin kytkettyjen ohjausjärjestelmien syötteiden generointiin. Ohjelma käynnistyy Test Managerin käskystä ja ottaa käynnistyshetkellään vastaan testisekvenssin.

Testisekvenssinä käytetty skriptikieli on normaalin lausekielen kaltainen, mutta komennot ovat erityisesti tarkoitettu työkoneen liikkeiden hallintaan. Kielen rakenne on pyritty pitämään yksinkertaisena niin, että se olisi mahdollisimman helppokäyttöinen ja helppo oppia. Kielen rakenne mahdollistaa makrojen tekemisen, jonka seurauksena yhdellä käskyllä voidaan suorittaa monimutkaisia toimintoja. Tämä helpottaa myös kielen käyttöä, sillä käyttäjän ei tarvitse tietää miten makrot toimivat sisäisesti vaan riittää että se toimii halutusti.

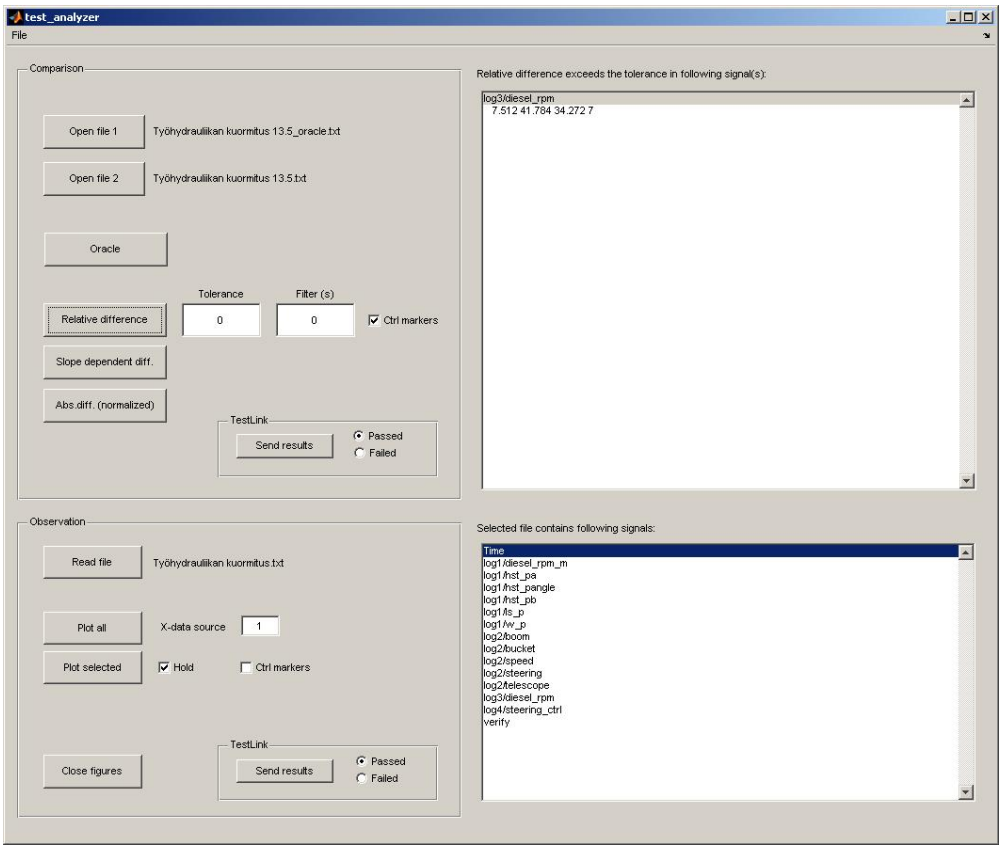
Kielen avulla voidaan vaikuttaa joko suoraan tiettyyn ohjaussignaaliin tai epäsuorasti ohjausjärjestelmän takaisinkytkentöihin ajamalla simuloitu kone tiettyyn asemaan tai tilaan. Toisin sanoen kieli voi sisältää säätimiä, joita tarvitaan luvussa 2.4.2 esitettyjen testitapausten generointimenetelmien käyttämiseen. Kielessä käytetyt muuttujat saadaan kahdesta tiedostosta, joihin on kirjoitettu rajapinnat ohjattavaan järjestelmään. UDP.ini tiedostossa on kerrottuna UDP-pakettien kautta välitettävät muuttujat ja parameters.ini tiedostossa on kerrottu takaisinkytkentöjen avulla säädettävät muuttujat.

3.1.4. Analysointi

Testien suorittamisen jälkeen testeistä on kerättynä suuri määrä dataa. Datan analysointiin ja visualisointiin kehitettiin GIMsim Test Analyzer ohjelma, josta käytetään vastaisuudessa nimitystä Test Analyzer. Sen avulla voidaan piirtää signaaleja ja verrata niitä jo olemassa oleviin tietoihin siitä, miten signaalin olisi pitänyt käyttäytyä. Vertailusignaaleja on mahdollista saada joko aikaisemmin suoritetuista testeistä tai suorittamalla testissä käytetyt ohjaukset ohjausjärjestelmän kuvausta vastaan. Tällöin saadaan toinen lokitiedosto, joka kuvaa miten ohjausjärjestelmän olisi pitänyt toimia saaduilla ohjauksilla.

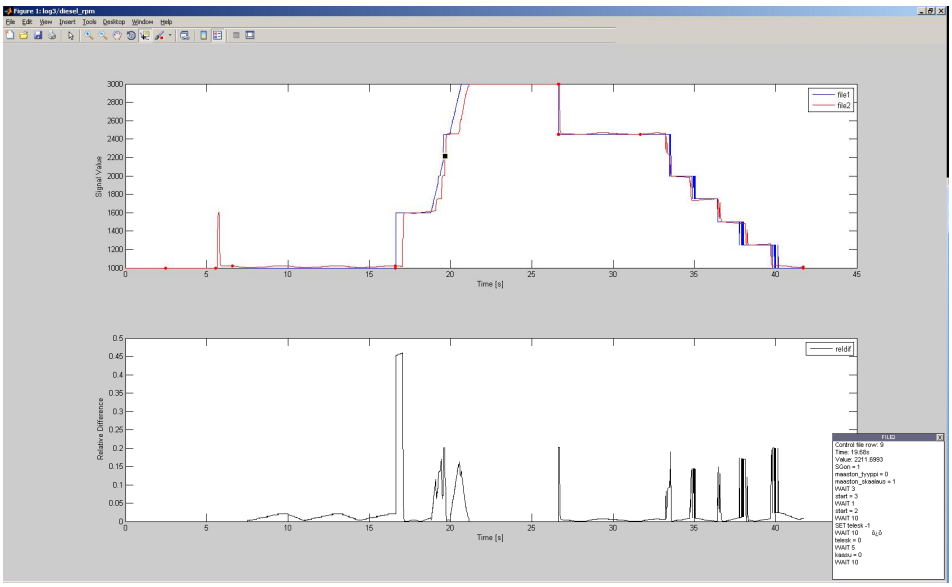
Ohjausjärjestelmän kuvaus kirjoitetaan Matlab-koodina, jonka Test Analyzer osaa suorittaa. Tällöin otetaan lokitiedoston kopiosta tietyltä ajanhetkeltä ohjaussignaalit ja annetaan ne ohjausjärjestelmän kuvaukselle. Se laskee saaduista arvoisa lokitiedostossa olevat muut arvot ja tallentaa ne vanhojen tietojen tilalle. Menetelmä on sama joka kuvattiin lyhyesti luvun 2.7 lopussa.

Signaalien vertailuun on käytettävissä kolme eri menetelmää, jotka ovat relative difference, slope dependent difference ja absolute difference. Muita menetelmiä on mahdollista lisätä tarpeen mukaan. Kuvassa 23 on Test Analyzer ohjelman pääikkuna.



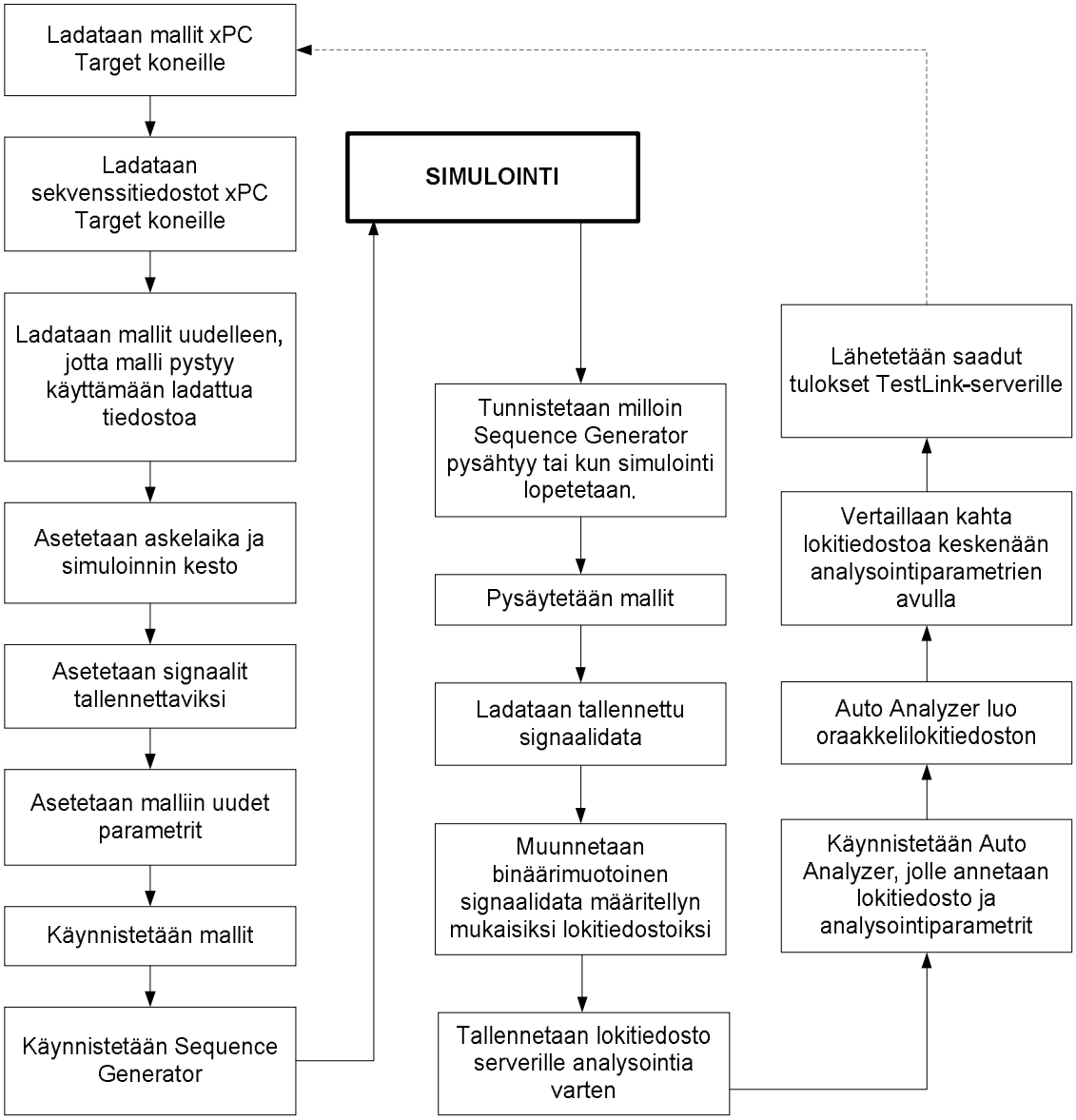
Kuva 23. GIMsim Test Analyzer-ohjelman pääikkuna.

Ohjelman yläosassa on signaalien vertailusta vastaava osio. Siihen pystyy avaamaan kaksi lokitiedostoa, joita pystytään vertailemaan halutuilla menetelmillä ja parametreilla. Listaan tulee näkyviin signaalit, joista löytyi eroavaisuuksia. Näitä kahta signaalia pääsee tarkastelemaan lähemmin tuplaklikkaamalla signaalia. Tällöin aukeaa kuvassa 24 esitetty ikkuna, jossa vertaillut signaalit näkyvät piirrettynä päällekkäin. Ikkunan alaosassa näkyy signaaleista löydetyt eroavaisuudet.



Kuva 24. Signaalien vertailunäkymä.

Ohjelman pääikkunan alaosassa näkyy avatun tiedoston sisältämät signaalit. Niitäkin pääsee tarkastelemaan lähemmin tuplaklikkaamalla haluttua signaalia. Analysointi voidaan myös suorittaa automaattisesti kun testi on suoritettu. Tällöin käytetään Auto Analyzer ohjelmaa, joka vastaa toiminnallisuudelta Test Analyzeriä, mutta ei sisällä käyttöliittymää. Kuvassa 25 on esitetty testausjärjestelmän suoritussekvenssi kokonaisuudessaan. Siinä kuvataan testauksen suoritus, kun testejä suoritetaan automaattisesti ja analysointiin käytetään Auto Analyzer ohjelmaa.



Kuva 25. Testien suoritussekvenssi IHA:n testausjärjestelmässä.

Vasemman puoleinen osa sekvenssistä suoritetaan ennen simulointia, näistä toiminnoista vastaa Test Manager. Keskimmäinen osa kuvaa simuloinnin jälkeen tapahtuvia toimintoja, joista vastaa myös Test Manager. Oikean puoleinen osio kuvaa Auto Analyzerin toimintoja. Ylhäällä oleva katkoviiva kuvaa siirtymistä seuraavaan testitapaukseen ja syklin aloittamista alusta.

4. GIMSIM TEST MANAGER OHJELMISTON KEHITYS

GIMsim Test Manager on ohjelma, joka vastaa automaattisen testauksen konseptin toteutuksessa testauksen hallinnoinnista. Kyseinen ohjelmisto esiteltiin lyhyesti luvussa 3, mutta tässä kappaleessa perehdytään tarkemmin sen kehitykseen, toimintaan ja rakenteeseen. Luvun tarkoituksena on antaa kuvaus siitä miten xPC Target ympäristöön on mahdollista kehittää ohjausjärjestelmien automaattinen testaustyökalu. Ohjelman kehityksessä on pyritty noudattamaan luvussa 2 kuvattuja tapoja ja menetelmiä. Ohjelmointikieleksi valittiin C++, koska reaaliaikasimulaattoreiden rajapinta, xPC Target API, oli ohjelmoitu C-kielellä.

Ohjelman toiminnot antavat mahdollisuuden käyttää sitä ohjelmiston kehityksen eri vaiheissa. Tämän mahdollistaa se, että signaalien generointi ohjausjärjestelmälle ja mallille voidaan toteuttaa joko simulaattorilla ajettavan tiedoston avulla tai GIMsim Test Generator ohjelman avulla simulointiympäristön ulkopuolelta. Regressio- ja back-to-back-testauksessa käytettäisiin ensimmäistä tapaa ja muuten jälkimmäistä tapaa. Ohjelmaa voidaan myös käyttää itsenäisesti tai liittää se osaksi TestLink-ohjelmaa, jolloin TestLinkissä hoidetaan testien luominen ja varastointi.

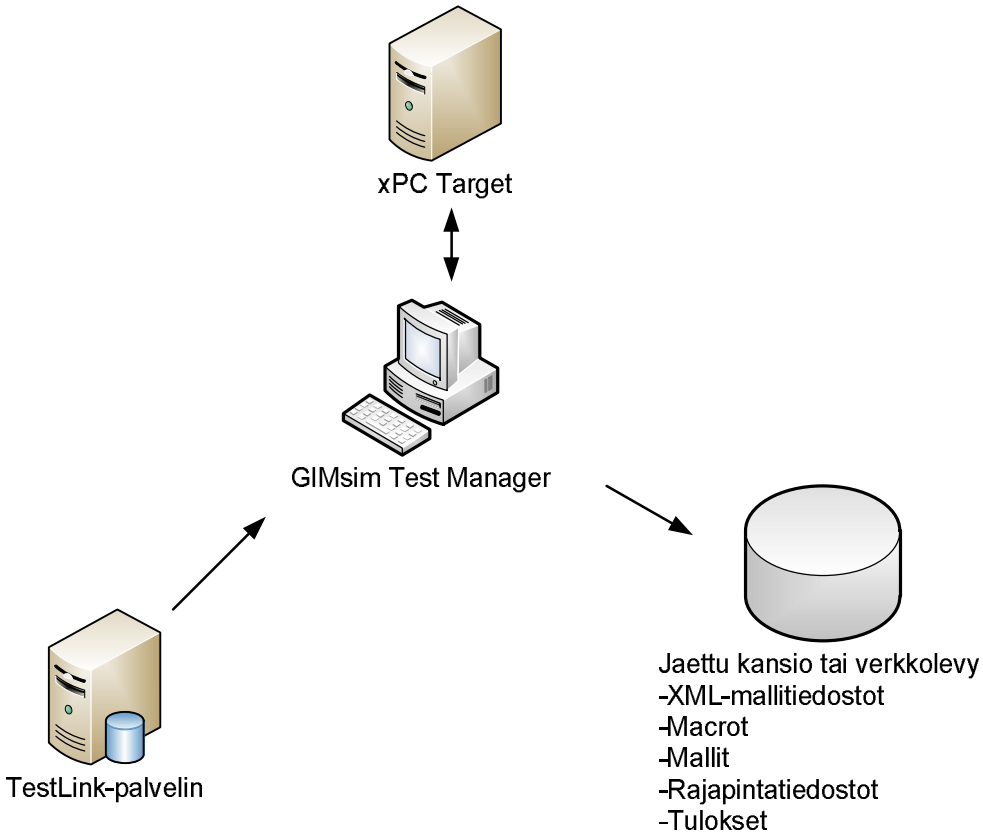
4.1. Toteutuksen keskeiset ratkaisut

Ohjelman kehityksessä pyrittiin huomioimaan mahdollisimman paljon luvussa 2 esiteltyjä ideoita ja menetelmiä. Kaikkia asioita ei pystyttä toteuttamaan tai kokeilemaan simulointiympäristön rajoitteiden vuoksi tai ajan loppumisen takia. Ohessa on kuvattu ohjelman toteutuksen tärkeimpiä ratkaisuja.

4.1.1. Tietojen eriyttäminen

Luvussa 2.3 esitettiin miten eriyttämällä testijärjestelmän osia toisistaan voidaan parantaa järjestelmän uudelleenkäytettävyyttä ja ylläpidettävyyttä. Test Managerin ympärillä olevan järjestelmän havaintokuva 26 osoittaa, kuinka järjestelmän eri osat ja tiedot on eriytetty toisistaan. Test Managerin tapauksessa testitapaukset, mallit, mallien rajapinnat, makrot ja saadut tulokset on eriytetty toisistaan. Kaikki nämä tiedot on tallennettu erikseen jaettuun kansioon tai testitapauksen kohdalla TestLink-palvelimelle. TestLinkissä kuvataan testitapaus, johon sitten liitetään haluttu XML-mallitiedosto ja sekvenssi. XML-mallitiedostossa on koottuna mallit, niihin liittyvät alustustiedot ja rajapintatiedostot, yhteen XML-tiedostorakenteeseen testien luomisen helpottamiseksi.

XML-mallitiedosto on esitettynä liitteessä 3. Testeistä tallennetut lokitiedostot tallennetaan myös jaettuun kansioon.



Kuva 26. GIMsim Test Managerin ympärillä olevan järjestelmän havaintokuva.

Tietojen eriyttämisellä saavutetaan se hyöty, ettei samoja tiedostoja ja tietoja tarvitse kopioida jokaiseen testiin erikseen vaan riittää, että viitataan olemassa oleviin tiedostoihin. Tällöin myös tiedostoissa, kuten simulointimalleissa olevat virheet voidaan korjata yhteen tiedostoon, koska kaikki testitapaukset käyttävät samoja simulointimalleja.

4.1.2. Datan keräyksen toteutus

Luvussa 2.6 on esitetty erilaisia menetelmiä, joilla datan keräys on toteutettu. Test Manager toteuttaa datan keräyksen hyödyntämällä xPC Targetin tarjoamaa signaalien tallennusta, jossa halutut signaalit tallennetaan xPC Target-koneen muistiin. Koneen muistiin tallennetut tiedostot haetaan muistista Test Managerin toimesta ja tallennetaan ymmärrettävässä muodossa jaettuun kansioon.

Tämä toteutus valittiin, jotta ei aiheuteta ylimääräistä liikennettä verkkoon ja jotta ei rajoitettaisi tallennusmahdollisuutta vain verkon yli lähetettäviin signaaleihin. Etukäteen ei voi olla aina varma mitä signaaleja halutaan tutkia, joten tämä toteutus tarjoaa enemmän vapauksia testaajalle.

4.1.3. Tiedostojen hallinta

Tiedostot, joihin TestLinkissä viitattiin testitapausten määrittelyn yhteydessä, tulee olla määrätyissä kansiossa, jotta ohjelmisto löytäisi ne. Test Managerin alustustiedostossa on määrätty mistä kansiosta mitäkin tiedostoja etsitään. Sekvenssitiedostot, makrot, simulointimallit, XML-mallitiedostot ja rajapinnat määrittelevät .ini-tiedostot tulee sijaita niille määrätyissä kansiossa.

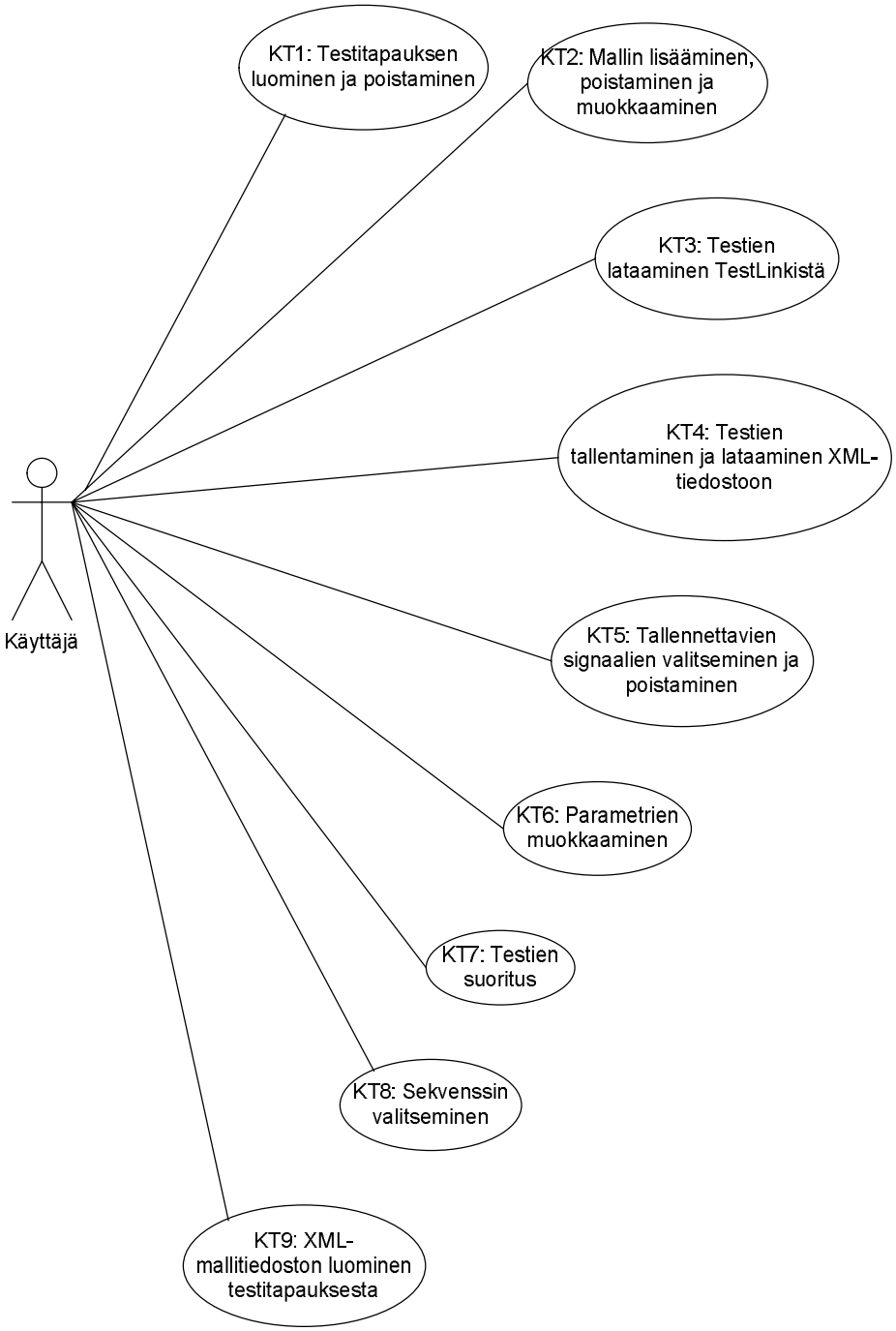
Tietojen säilytykseen on varattuna jaettu muistialue, kuten Windowsin jaettu kansio tai verkkolevy. Tällöin samaan tietoon pääsee käsiksi sekä TestLink serveriltä, että simuloinnin hallinta PC:ltä. Suositeltu kansiorakenne on kuvattuna liitteessä 2.

4.1.4. Matlab enginen käyttö

Test Manager käyttää myös XpcToSequence-ohjelmaa. Se kääntää tallennetun signaalitiedoston FromFile-lohkon hyväksymään muotoon. Kyseinen ohjelma on alun perin kirjoitettu Matlab m-fileksi, joka on sitten käännetty erilliseksi ohjelmaksi. Sen olisi voinut myös tehdä Matlab enginen kautta, mutta Test Managerista haluttu tehdä mahdollisimman ohjelmistoriippumaton.

4.2. Toiminnot

Test Managerin toiminnot on kuvattu käyttötapauskaaviolla, joka on esitetty kuvassa 27. Käyttäjäluokkia ohjelmassa on vain yksi ja käyttötapauksia yhdeksän. Käyttötapaukset kuvaavat kaiken tarpeellisen toiminnan, joiden avulla käyttäjä pystyy luomaan ja suorittamaan testejä.



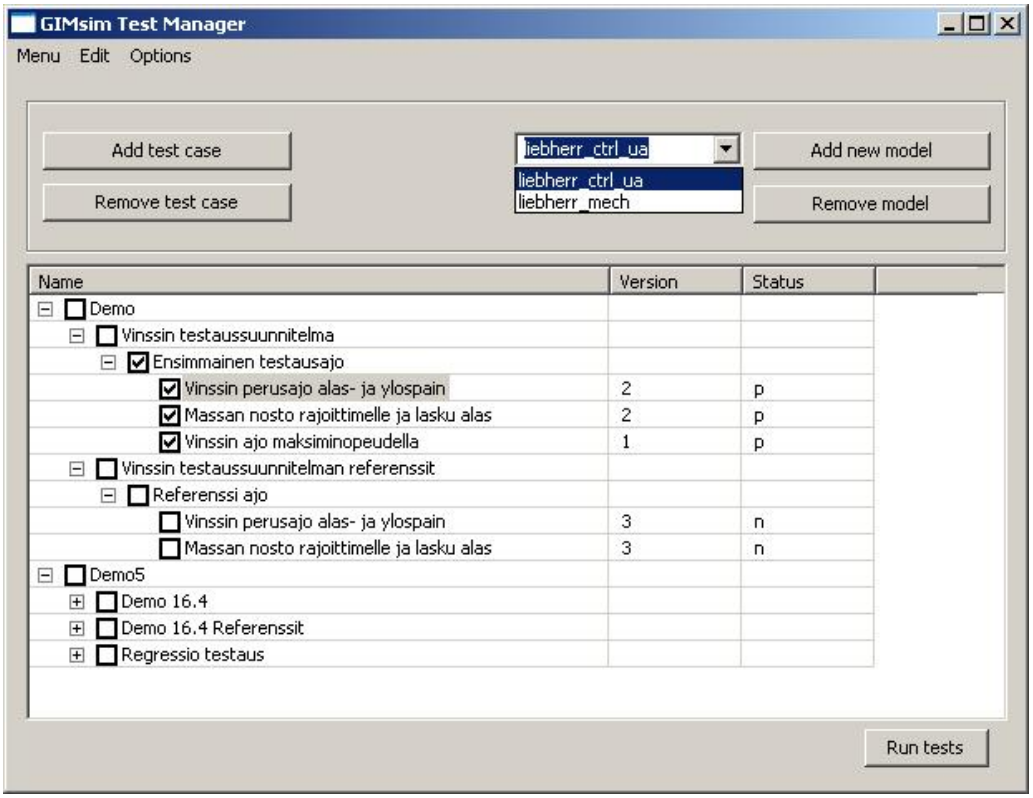
Kuva 27. Käyttötapauskaavio.

Käyttötapausten tarkemmat kuvaukset on esitetty liitteessä 5. Kuvassa näkyvien käyttötapausten lisäksi liitteessä on kuvattu kaksi TestLinkiin liittyvää käyttötapausta. Niiden kuvaaminen katsottiin tarpeelliseksi, jotta lukijalle tulisi selväksi miten ohjelmat liittyvät toisiinsa.

4.3. Käyttöliittymä

Pääikkunan, joka on kuvassa 28, tärkein toiminto on näyttää ohjelman sisältämät testitapaukset. Ne näytetään puulistassa, lajiteltuna projektin, testaussuunnitelman ja

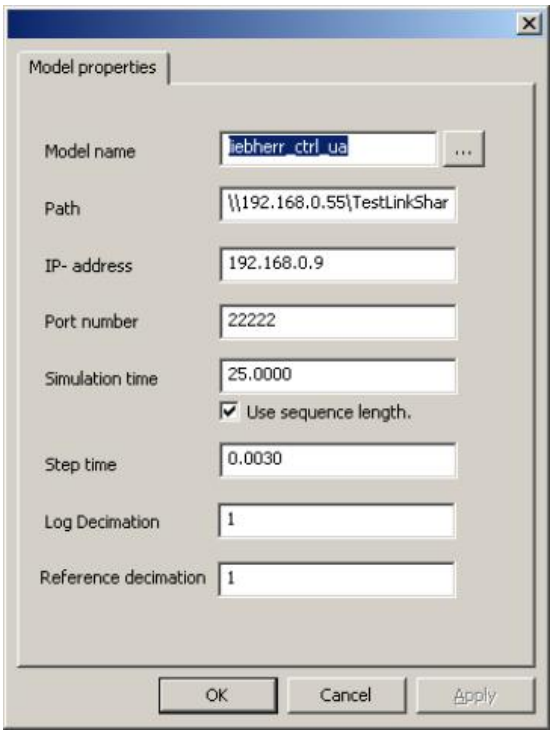
buildin avulla. Testitapausten yhteydessä näytetään versio ja suoritusstatus. Jos testit on ladattu XML-tiedostosta niin ne näkyvät yksittäin, eivätkä puulistassa. Puulistan tarkoituksena on näyttää testit samalla tavalla kuin ne on järjestetty TestLinkiin. Ylempänä samassa ikkunassa näytetään valitun testitapauksen mallit. Malleja voidaan lisätä ja poistaa sen vieressä olevista napeista.



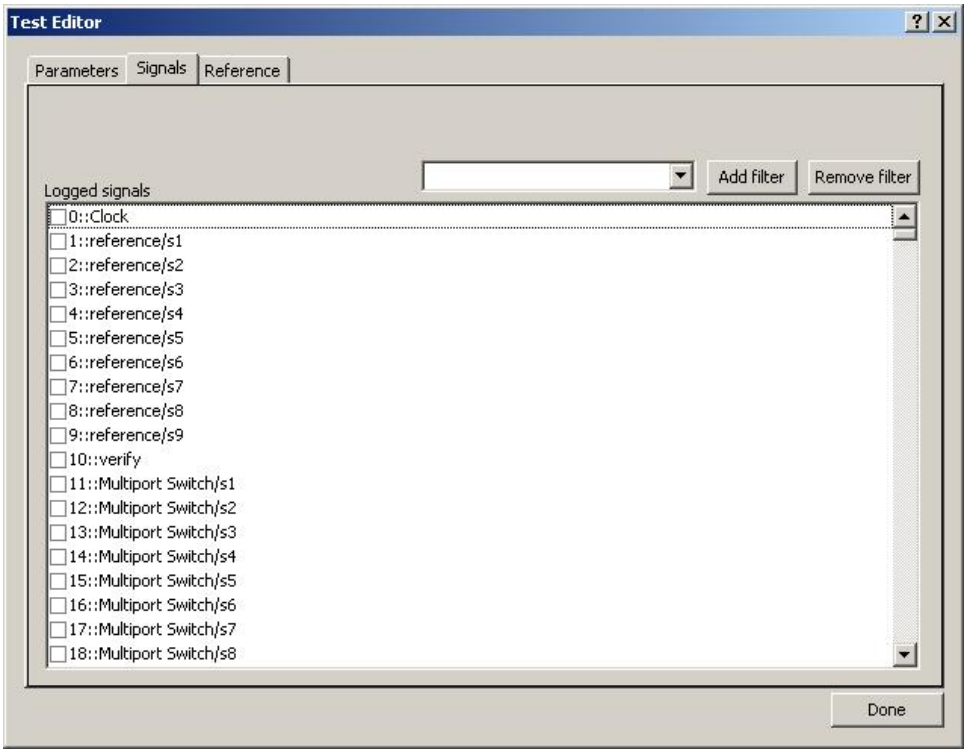
Kuva 28. Test Manager pääikkuna.

Menu-valikosta voidaan tallentaa ja ladata testejä XML-muotoisista tiedostoista. Lisäksi sieltä voidaan ladata testejä TestLinkistä ja tallentaa TestLinkin käyttämiä XML-mallitiedostoja.

Edit-valikosta pääsee muokkaamaan testitapauksen ja mallien parametreja. Model properties-valikosta aukeaa kuvan 29 mukainen ikkuna, jossa voi muokata mallin tietoja. Test editor-valikosta aukeaa kuvan 30 mukainen ikkuna, jossa voidaan muokata parametreja ja tallennettavia signaaleja erillisissä väli-ikkunoissa. Test files-valikosta voidaan valita testin aikana ajettava sekvenssitiedosto.



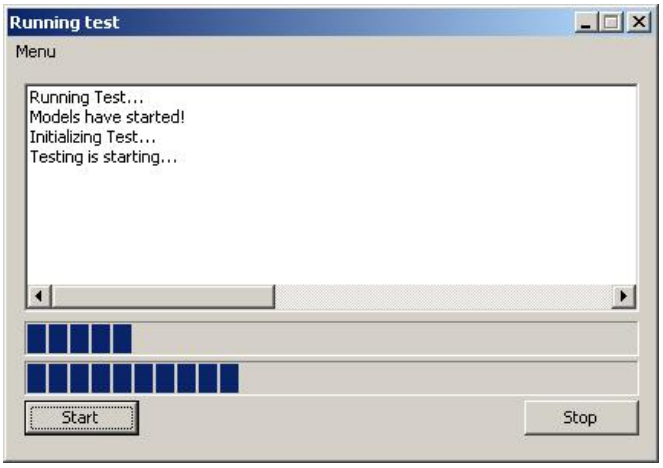
Kuva 29. Mallin tietojen muuttaminen.



Kuva 30. Signaalien valitseminen.

Kuvan 31 mukaisessa testin suoritus ikkunassa näytetään testauksen eteneminen sanallisesti ja simulaation käynnistyttyä myös ajallisesti. Testauksen voi käynnistää uudelleen alusta painamalla Start-nappia, tai pysäyttää kesken suorituksen painamalla Stop-nappia. Menu-valikosta saadaan valittua sekvenssien valinta ikkuna. Tällöin voidaan myös käynnistää Sequence Generator manuaalisesti. Tämä mahdollistaa

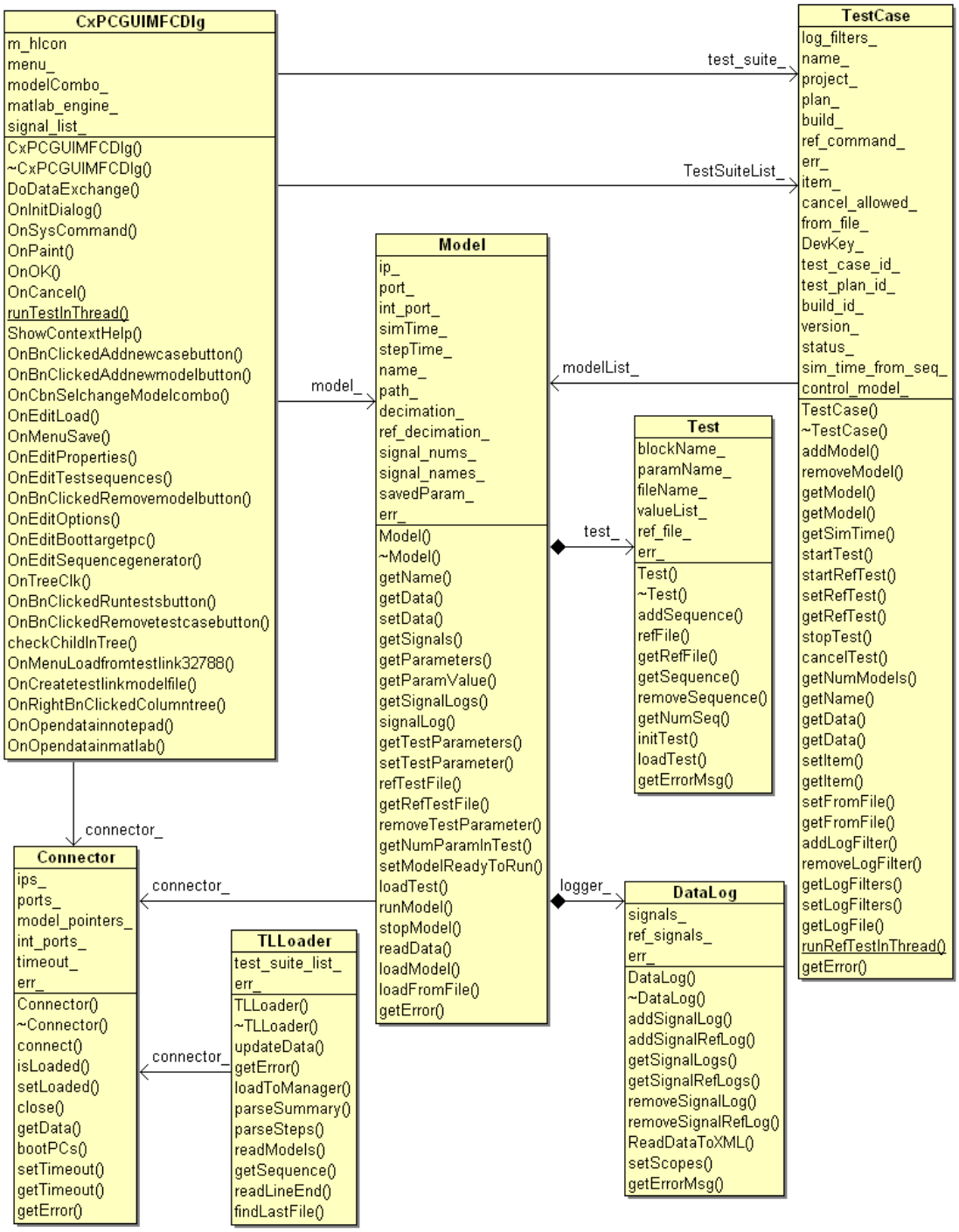
sekvenssien testaamisen ja niiden ajamisen peräkkäin ilman, että simulaattorit käynnistetään välillä uudelleen.



Kuva 31. Testin suoritus.

4.4. Ohjelmistorakenne

Tässä osassa kuvataan ohjelman rakennetta luokkakaavion avulla. Kuvassa 32 on esiteltynä ohjelman tärkeimmät luokat. Luokat ja niiden rajapinnat on esitelty omissa kappaleissa.



Kuva 32. Luokkakaavio.

Käyttöliittymä ikkunoista esitellään ainoastaan CxPCGUIMFCDlg-luokka joka on pääikkunan toteuttava luokka. Tämän luokkakaavion ulkopuolelle on jätetty muut käyttöliittymän ikkunat toteuttavat luokat. Niiden esittely olisi enemmänkin vaikeuttanut ohjelman rakenteen selvittämistä kuin helpottanut sitä, jonka takia kyseisten luokkien esittelystä luovuttiin. Kuvan 32 luokkakaavio kuvaa hyvin ohjelman toiminnallisen rakenteen.

Yhteisenä piirteenä kaikilla luokilla on virheiden hallinta. Tavoitteena oli, että virhetilanteiden hallinta voitaisiin pitää kunkin luokan sisäisenä asiana. Kun luokan

sisällä tapahtuu virhe, ilmoitetaan siitä funktiota kutsuneelle osalle palauttamalla bool tai int arvo, joka ilmoittaa virheen luonteen. Luokan sisäiseen err_-muuttujaan tallennetaan sitten virhettä kuvaava virheviesti, joka on haettavissa getError-funktion avulla.

4.4.1. CxPCGUIMFCDlg-luokka

Luokka toimii ohjelman pääikkunan toteuttavana luokkana. Sen sisäisinä muuttujina ovat testitapaukset, jotka kuvataan TestCase-olioina ja aktiivisena oleva malli, joka kuvataan Model-oliona. Matlabin engine on myös tallennettuna tänne, jotta samaa moottoria voidaan käyttää kaikissa luokissa. Myös Connector olio on kaikille malleille sama, jonka takia sekin on tallennettu tähän luokkaan. Luokan kautta toteutetaan kaikki pääikkunan toiminnot funktioina.

4.4.2. TestCase-luokka

Luokka toteuttaa testitapausta vastaavat tiedot ja toiminnallisuuden. Muuttujina on testitapaukselle annettavia parametreja kuten signaalien tallentamiseen käytettävät loki filtterit, analysoinnissa käytettävät parametrit sekä muut TestLinkissä määrättävät parametrit. TestLinkin kanssa yhteydenpidossa tarvittavat parametrit on myös tallennettu tänne.

Luokan sisältämiin malleihin päästään käsiksi getModel-funktioilla, jotka palauttavat viittauksen malliin. Mallin tietoja ei siis tarvitse päivittää TestCase-luokan läpi. Testien aloittamiseen ja lopettamiseen liittyvät funktiot on myös sisällytetty luokkaan.

4.4.3. Model-luokka

Luokka toteuttaa mallia ja xPC Target-konetta kuvaavat tiedot ja toiminnallisuuden. Luokan muuttujina ovat xPC Target-koneen yhteystiedot ja mallin ajamisessa käytettävät tiedot.

Funktioiden avulla toteutetaan yhden mallin ajamiseen ja alustamiseen tarvittava toiminnallisuus. Luokan funktioiden kautta ollaan yhteydessä sen omistamiin Test- ja DataLog-luokkiin, joissa osa näistä toiminnallisuuksista toteutetaan.

4.4.4. Test-luokka

Luokka toteuttaa malliin liittyvien testien parametrien ja sekvenssien hallinnoinnin. Sen muuttujina ovat malliin muutettavat parametrit ja sekvenssin nimi, joka mahdollisesti ajetaan.

Luokan sisältämät funktiot toteuttavat parametrien muuttamiseen liittyvät toiminnot. Testien ajonaikaista hallinnointia ei suoriteta eikä esimerkiksi sekvenssejä käynnistetä tässä luokassa, vaan sekvenssin nimi ja sijainti annetaan tarvittaessa Model-luokalle, jonka vastuulla on ajonaikainen hallinta.

4.4.5. DataLog-luokka

Tämä luokka toteuttaa signaalien tallentamisesta vastaavat toiminnot ja tiedot. Sen muuttujina ovat signaalit, joita pitää mallin suorituksen aikana tallentaa.

Luokan funktiot vastaavat signaalien tallentamisen alustamisesta ja testin loputtua myös tallennuksen lopettamiseen liittyvistä toiminnoista. Tallennettu data muutetaan liitteessä 4 kuvatun lokitiedoston kaltaiseksi, kun dataa haetaan malleista.

4.4.6. Connector-luokka

Luokka vastaa yhteyksien hallinnasta xPC Target-koneille. Se sisältää tiedot kaikista verkossa olevista koneista, joihin on oltu yhteydessä. Luokka tietää myös, mikä malli on ladattuna millekin koneelle, jolloin vältytään mallien ylimääräisiltä latauksilta.

Funktioiden avulla voidaan ottaa tai purkaa yhteys tiettyyn xPC Target-koneeseen ja tarkistaa mitkä mallit koneelle on ladattu. Virhetilanteita voi ilmaantua, jos xPC Target-koneille ei saada yhteyttä tai koneet ovat kaatuneet. Tällöin ainoa mahdollisuus virhetilanteista toipumiseen on käynnistää koneet manuaalisesti uudelleen.

4.4.7. TLLoader-luokka

Luokka vastaa yhteydestä TestLinkiin ja erityisesti testien hakemisesta. Muuttujina ovat testien haussa päivitettävä lista testitapauksista ja virhemuuttuja. Loput tiedot saadaan TestLinkistä.

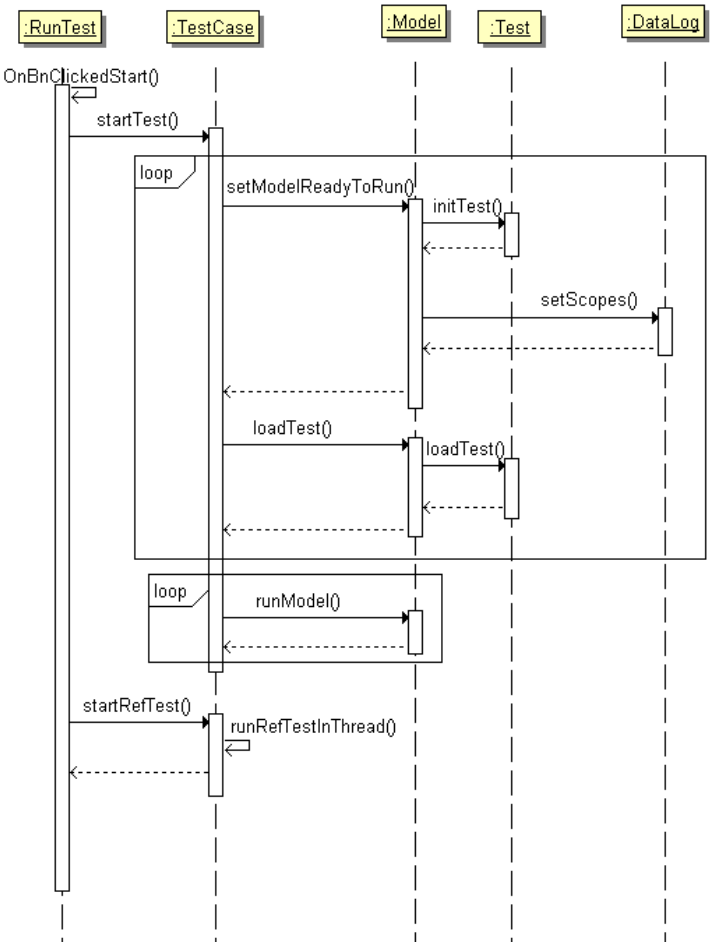
Funktiot on jaettu toiminnallisuuksiin, joissa eri tietoja etsitään TestLinkistä saadusta datasta. ParseSummary ja parseSteps vastaavat testien tietojen erittelystä, kun tekstin joukosta etsitään haluttuja muuttujia, kuten analysoinnissa käytettäviä parametreja. Ulospäin luokasta näkyy ainoastaan updateData ja getError-funktiot. Loput funktiot ovat sisäisiä.

4.5. Toiminta ja tilat

Ohjelman suorituksesta kuvataan vain toteutuksen kannalta keskeisimpiä toimintoja. Kunkin toiminnon kuvaamiseen käytetään UML-sekvenssikaavioita. Niissä kuvataan toiminnon sisältämiä funktiokutsuja. Luokkien funktiot on pyritty nimeämään loogisesti niin, että niiden nimistä selviää myös käyttötarkoitukset.

4.5.1. Testin käynnistys

Kun käyttäjä on valinnut halutut testitapaukset suoritettavaksi, aloitetaan kuvan 33 mukaisen sekvenssin suoritus.



Kuva 33. Testin käynnistyksen sekvenssikaavio.

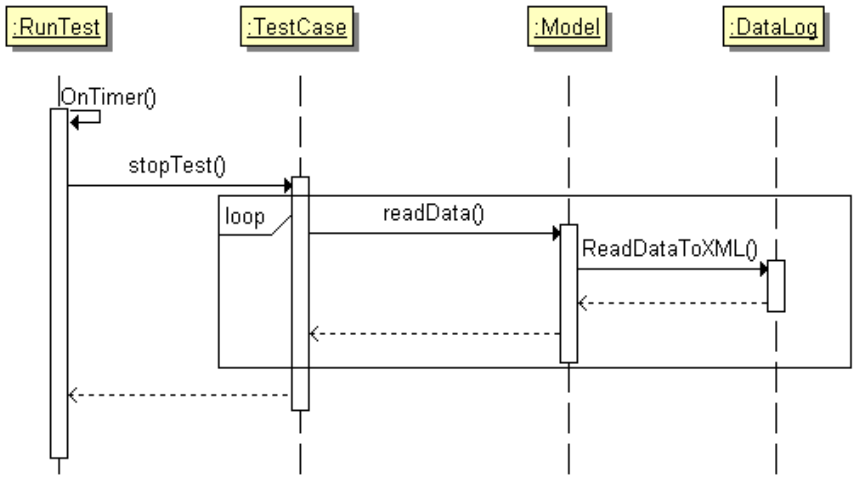
Testien suoritus aloitetaan kun käyttäjä painaa Start-nappia. Sekvenssi alustaa reaaliaikakoneet, mallit, sekvenssin ja datan tallennuksen. Kun nämä tiedot on ensin alustettu, mallit käynnistetään mahdollisimman samanaikaisesti. Tämän jälkeen käynnistetään sekvenssien suorituksesta vastaava Test Generator, jos se on valittu testin suoritustavaksi. Kyseinen ohjelma ajetaan erillisessä säikeessä, jotta ohjelmien suoritus olisi mahdollisimman sujuvaa.

Kaikkien mallien käynnistyttyä ja sekvenssin ajon käynnistyttyä asetetaan ajastimet, joilla säädellään testin suorituksen lopetusta. Jos malleille on asetettu simulointiajat, alustetaan ajastimet näillä tiedoilla. Jos taas simuloinnin kesto määritettään Sequence Generator ohjelman suorituksesta, alustetaan ajastin laukaistavaksi sekunnin välein, jolloin tarkastetaan onko kyseisen ohjelman status. Testin lopetuksen sekvenssi on kuvattuna seuraavassa luvussa.

4.5.2. Testin lopetus ja tiedonhaku

Ohjelma päättelee testin suorituksen lopetuksen, joko simulointiajasta tai Sequence Generator-ohjelman lopetuksesta. Simulointiajan ja sekvenssin pituuden kuluminen määritellään ajastimen avulla. Sequence Generator taas suoritetaan erillisessä säikeessä, jonka suoritusta tarkkaillaan globaalilla muuttujalla. Tätä muuttujaa taas tarkastellaan

ajastimella sekunnin välein. Kun jokin näistä ehdoista täyttyy, niin aloitetaan kuvan 34 mukaisen sekvenssin suoritus.

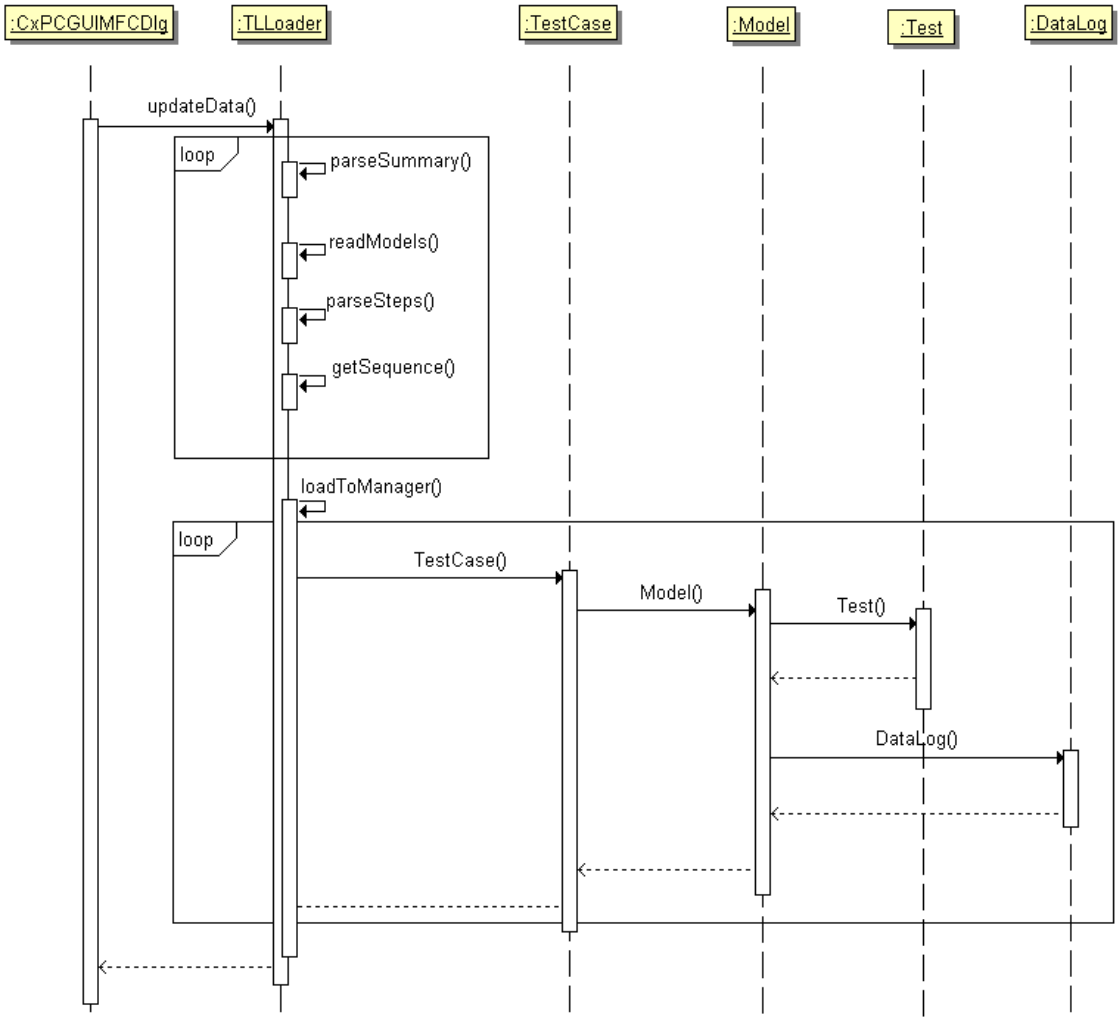


Kuva 34. Testin lopetuksen sekvenssikaavio.

Testin loputtua pysäytetään ensiksi simulointimallit. Tämän jälkeen haetaan tallennettu signaalidata. Signaalidata on binäärimuodossa, jonka takia se muutetaan vielä luettavampaan muotoon. Muunnettu lokitiedosto on kuvattuna liitteessä 4.

4.5.3. Testien haku TestLinkistä

Testitapaukset on kirjoitettuna TestLinkiin. Kun käyttäjä painaa Load from TestLink-nappia, aloitetaan kuvassa 35 olevan sekvenssin suoritus.



Kuva 35. Testien haku TestLinkistä sekvenssikaavio.

Sekvenssin alussa ladataan updateData-funktiossa TestLinkissä olevat testitapaukset. Tällöin testitapaukset ovat vielä tekstinä, josta pitää etsiä kaikkia tarvittava data, jolloin voidaan generoida testitapaukset. Jokaisen ladatun testitapauksen tiedot tutkitaan läpi parseSummary- ja parseSteps-funktioissa. ReadModels-funktiossa luetaan XML-mallitiedosto, joka testiin on liitetty.

Kun kaikki tarvittava data on saatu etsittyä, luodaan testitapaukset. Sekvenssikaaviossa ei ole huomioitu kaikkia luokkien alustamiseen liittyviä funktiokutsuja, koska sitä ei katsottu tarpeelliseksi. Testitapausten luomisen jälkeen päivitetään vielä taulukko pääikkunan taulukko, jossa testitapaukset näytetään.

4.6. Kehitysympäristö

Ohjelman kehitysympäristönä käytettiin Microsoft Visual Studio 2005:a. XML-tiedostojen lukemiseen ja kirjoittamiseen käytettiin ilmaista Chilkat Software XML C++ pakettia. Se on vapaan lähdekoodin projekti, joka tarjoaa kirjaston XML tiedostojen lukemiseen ja kirjoittamiseen [15]. XML-RPC pohjaiseen tiedonsiirtoon käytettiin myös vapaan lähdekoodin kirjastoa nimeltä XML-RPC for C and C++ [16]. Matlabin

käytössä oleva versio määräsi xPC Target API:n käytön. Ohjelman kehityksen alussa oli käytössä versio 3.5, mutta se päivitettiin myöhemmin versioon 4.0. Taulukkoon 1 on kerätty ohjelmiston kehitystyössä käytettyjä ohjelmia ja niiden versioita.

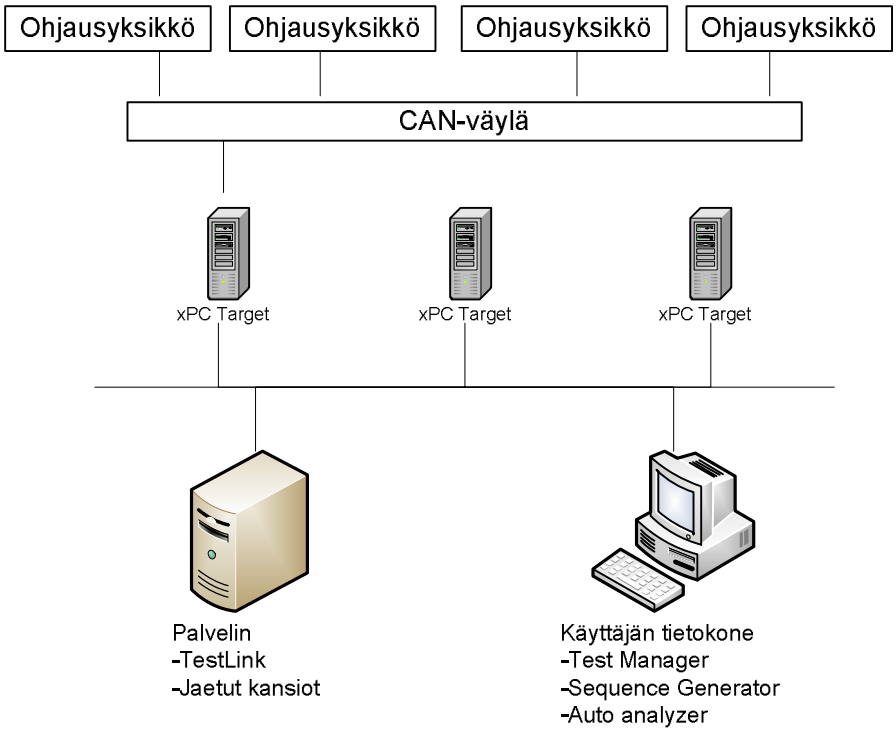
Nimi	Versio
Microsoft Visual Studio 2005	8.0.50727.762
Chilkat Software XML C++	8.0
XML-RPC for C and C++	1.16
xPC Target API 4.0	4.0

Taulukko 1. Kehitysympäristö.

5. CASE AVANT

Automaattisen testauksen konseptia testattiin IHA:ssa olevalla Avant-simulaattorilla. Avant on Avant Tecno Oy:n valmistama pienkuormaaja. Sitä käytetään normaalisti kevyissä maansiirtotöissä, maataloudessa ja viherrakentamisessa. Se valittiin Case kohteeksi, sillä Avant HIL-simulaattori oli valmiiksi käytettävissä hydrauliikan ja automatiikan laitoksella. Simulaattoriin oli mallinnettu Avant-kone kokonaisuudessaan Simulinkin avulla. Reaaliaikajärjestelmänä simulaattorissa oli MathWorksin xPC Target. Avantin ohjausjärjestelmä on rakennettu uudestaan IHA:n laitoksella tutkimus tarkoituksiin. Siitä testattiin ohjausjärjestelmän osaa, joka vastaa ajovoimansiirron ohjauksesta.

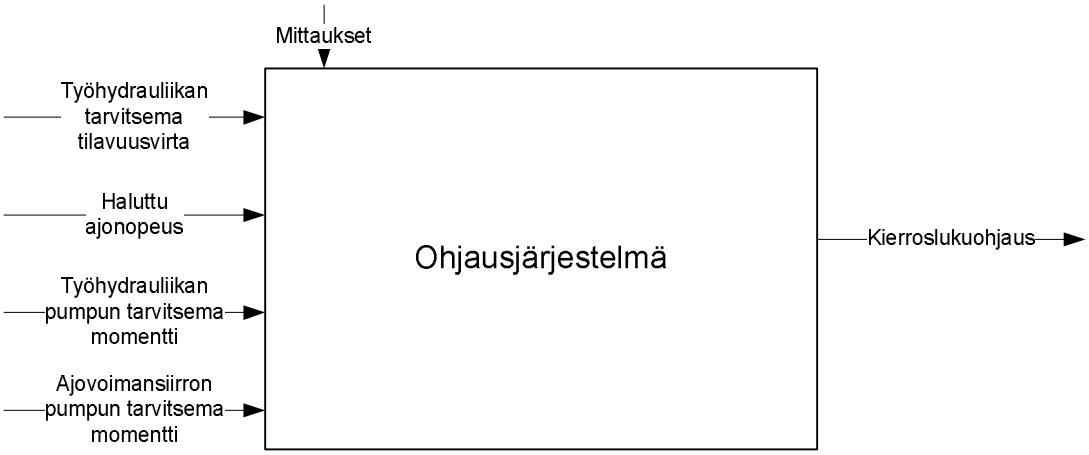
Valmis testausjärjestelmä, joka on esitetty kuvassa 36, koostui kolmesta xPC Target-simulaattorista, neljästä Epec Oy:n valmistamasta ohjausyksiköstä ja kahdesta Windows XP-tietokoneesta. Simulaattorit, palvelin ja käyttäjän tietokone oli liitetty yhteen lähiverkon välityksellä. Ohjausyksiköt olivat kiinni yhdessä xPC Target-simulaattorissa CAN-väylän välityksellä. Palvelimelle asennettiin Apache-serveri, TestLink-ohjelmisto ja sinne luotiin Windowsin jaettu kansiorakenne, joka on kuvattu liitteessä 2. Käyttäjän omalle tietokoneelle asennettiin Test Manager, Test Generator ja Test Analyzer ohjelmat. Tämän lisäksi käyttäjän tietokoneella oli käynnissä visualisointi, josta voitiin seurata Avant-kuormaajan toimintaa.



Kuva 36. Testausjärjestelmän kokoonpano.

5.1. Testien alustus

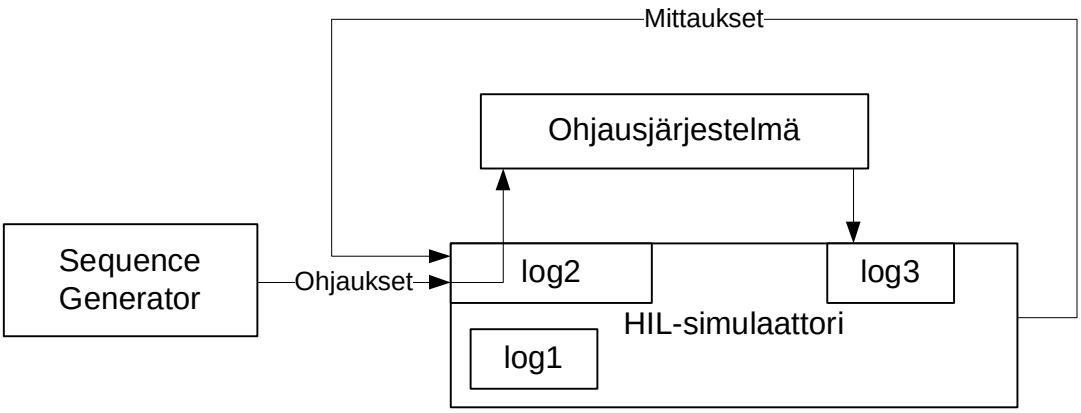
Testauksen suunnittelu aloitettiin kirjoittamalla ajovoimansiirron ohjausjärjestelmän kuvaus Matlab-koodina, joka toimii testauksen analysoinnissa vertailujärjestelmänä. Se tehtiin ohjausjärjestelmän määritysten pohjalta. Kuvassa 37 on kuvattu ohjausjärjestelmään tulevat sisääntulot ja näiden perusteella laskettu haluttu ulostulo. Kuten kuvasta näkyy, ohjausjärjestelmän ainoa ulostulo oli dieselmoottorilta haluttu kierrosluku. Tämä oli ainoa muuttuja jota tarkasteltiin, kun myöhemmin analysoitiin testin toimivuutta. Sisääntuloina ohjausjärjestelmään tuli erinäisiä mittaustuloksia sekä muilta ohjausjärjestelmän osilta tulevia ulostuloja. Ainoastaan haluttu ajonopeus saatiin suoraan Sequence Generator ohjelmalta ohjauksena.



Kuva 37. Ohjausjärjestelmän sisään- ja ulostulot.

Simulointimalleihin lisättiin sopivat UDP-rajapinnat, jotta simulaattoria pystyttiin ohjaamaan Sequence Generator-ohjelmalla. Nämä rajapinnat kuvattiin UDP.ini ja parameters.ini nimisiin tiedostoihin. UDP.ini-tiedostossa kuvataan mallin UDP-rajapinnat lajiteltuna sisääntuleviin ja ulostuleviin signaaleihin porteittain. Parameters.ini tiedostossa kuvataan signaalien riippuvuuksia, jolloin mallin ulostulo pystytään muuttamaan halutuksi muuttamalla siihen vaikuttavaa sisääntuloa. Lisäksi mallissa olevia signaaleja järjesteltiin uudelleen niin, että kiinnostavat signaalit kerättiin yhteen lohkoon. Näin ne voitiin löytää kyseisen lohkon nimen perusteella helpommin, jolloin myös niiden tallentaminen Test Manager-ohjelmalla helpottui. Tallennettaviin signaaleihin lukeutui kaikki ohjaus- ja mittaussignaalit, joita ohjausjärjestelmä tarvitsi ja kierroslukuohjauksen signaali. Kuvassa 38 on esitetty periaatekuva miten tallennettavat signaalit on jaoteltu simulaattorissa toimivassa simulointimallissa.

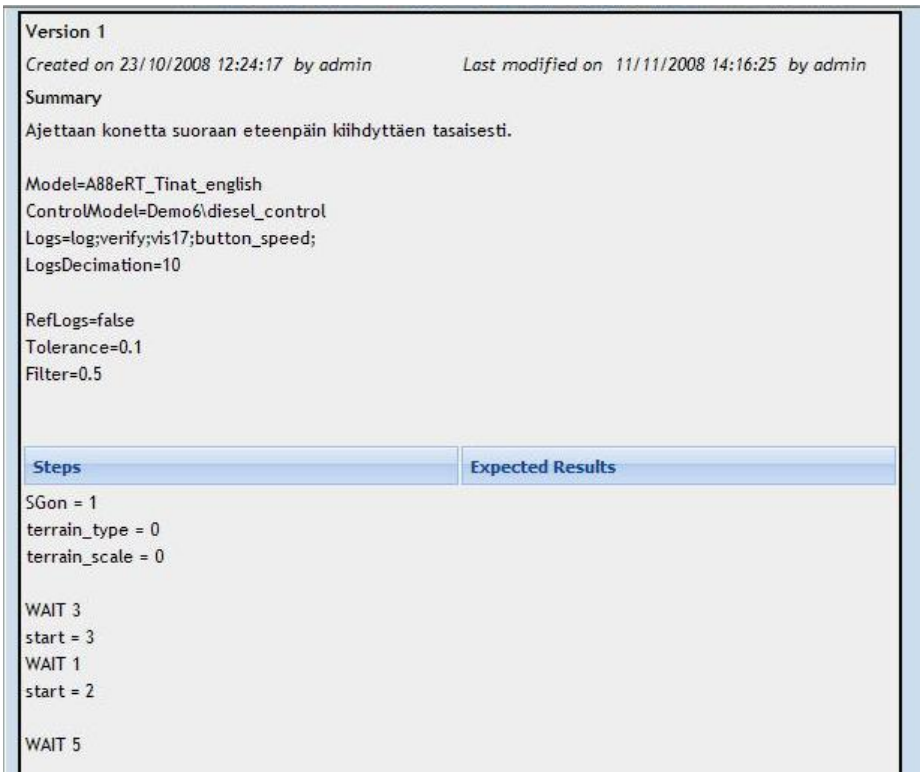
Log2 lohkoon on kerätty ohjausjärjestelmän käyttämät sisääntulosignaalit ja log3 lohkoon on kerätty ohjausjärjestelmän ulostulosignaalit. Log1 lohkoon on tallennettu simulaattorin sisäisiä muuttujia testeissä löytyvien virheiden korjaamisen helpottamiseksi. Tallentamalla näiden kahden lohkon signaalit saadaan kaikki tarvittava tieto ohjausjärjestelmän kuvauksen ajamiseen ja testien analysointiin.



Kuva 38. Simulaattorin ohjaus- ja mittaussignaalit sekä niiden tallentaminen.

Mallien tiedot ja .ini-tiedostot koottiin yhteen XML-mallitiedostoon, jonka esimerkkitoteutus on kuvattuna liitteessä 3. Tämän tiedoston avulla voidaan alustaa haluttu testi, ilman että kaikkia tietoja tarvitsee syöttää erikseen. Tiedosto rakennettiin Test Manager-ohjelman avulla.

Kun tarvittavat tiedostot olivat valmiina, alettiin kasata testitapauksia. Testitapauksia varten rakennettiin TestLinkiin mallipohja, joka on kuvassa 39. Siinä voitiin käyttää tulevien testitapausten kirjoituksessa. Siihen oli valmiiksi kirjoitettu testin tietoja kuten XML-mallitiedosto, ohjausjärjestelmän kuvaustiedosto, tallennettavat signaalit jne. Myös sekvenssin alku, jossa alustettiin simulaattorin alkutila, kirjoitettiin valmiiksi.


Version 1

Created on 23/10/2008 12:24:17 by admin *Last modified on 11/11/2008 14:16:25 by admin*

Summary

Ajettaan konetta suoraan eteenpäin kiihdyttäen tasaisesti.

Model=A88eRT_Tinat_english
ControlModel=Demo6\diesel_control
Logs=log;verify;vis17;button_speed;
LogsDecimation=10

RefLogs=false
Tolerance=0.1
Filter=0.5

Steps	Expected Results
SGon = 1 terrain_type = 0 terrain_scale = 0 WAIT 3 start = 3 WAIT 1 start = 2 WAIT 5	

Kuva 39. TestLinkin mallipohja johon on kirjoitettu testissä käytettävät perustiedot.

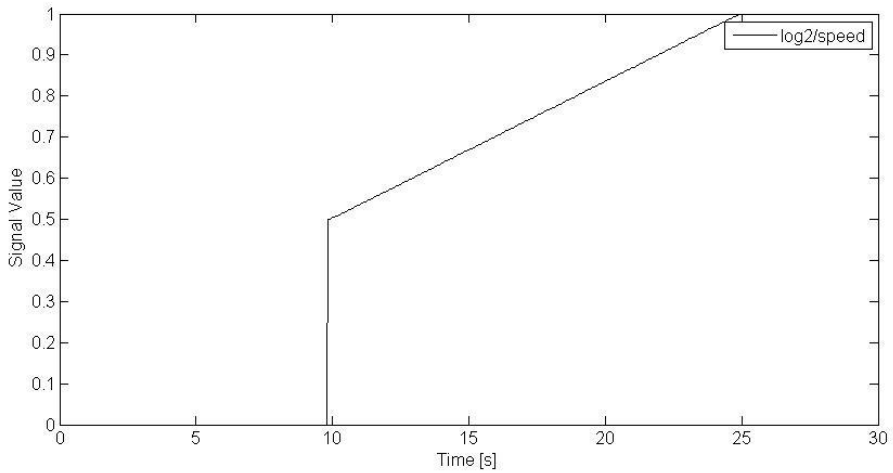
5.2. Testien määritys

Testien generointiin ei ollut käytettävissä automaattisia työkaluja, joten testitapaukset laadittiin käsin. Koska tavoitteena ei ollut testata koko ohjausjärjestelmän toimintaa, vaan testata automaattisen testauksen toimivuutta ja sen tuomia hyötyjä, testitapausten määrä rajattiin kolmeen. Nämä testit valittiin koska ne tuovat esiin automaattisen testauksen hyödyt testien suorituksen tarkkuudessa ja toistossa.

Testitapaukset kirjoitettiin TestLinkiin luodun valmiin mallipohjan päälle, jolloin testeihin tarvitsi kirjoittaa ainoastaan käytettävä sekvenssin ja analysoinnissa käytettävät toleranssit. Sopivien toleranssien arvot riippuvat paljon kyseisestä testistä ja näiden löytäminen voi olla joskus hankalaa. Jos käyttää liian tiukkoja arvoja, löytyy analysoinnissa virheitä jotka eivät oikeasti ole virheitä. Tämä on kuitenkin pienempi paha kuin virheiden löytymättä jääminen, joten on turvallisempaa käyttää hieman tiukempia toleranssin arvoja kuin löysempiä. Seuraavassa on kuvattuna testitapaukset ja niiden käyttämät sekvenssit.

Testitapaus 1: Ramppivaste

Ramppivasteen ajaminen antaa kuvan kaasun painamisen vaikutuksesta nopeuteen. Tällä testataan, että vaste on halutun funktion muotoinen. Testitapaus valittiin koska näin demonstroidaan automaattisen testauksen hyötyä tarkkuutta vaativassa testauksessa. Ramppi pystytään ajamaan tarkasti, ilman häiriöitä. Manuaalisella testauksella kyseisen testin suorittaminen olisi mahdotonta, ihmisestä johtuvan virheen vuoksi. Kuvassa 40 on esitettynä testissä tuotettu ohjaussignaali. Signaali on tallennettu testauksen aikana, kuvassa 38 näkyneestä log2 lohkosta

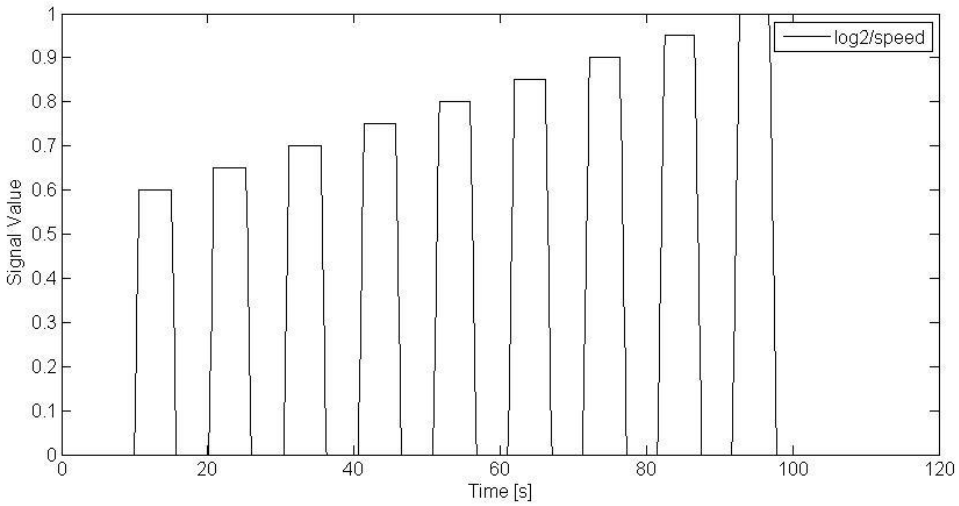


Kuva 40. Testitapaus 1:n testisekvenssi.

Testitapaus 2: Ajaminen eteen sykleittäin, eri kaasun arvoilla

Tässä testissä ajetaan konetta eteenpäin niin, että nostetaan ajamisessa käytettävänä kaasun painalluksen arvoa sykleittäin kunnes ajetaan täydellä kaasulla. Jokaisen kiihdytyksen välissä pysäytetään kone ja kiihdytetään taas uudelleen haluttuun

nopeuteen. Tässä testissä saadaan samankaltainen kuvaaja kuin ramppivasteen tapauksessa, mutta testi on toteutettu eri tavalla. Testitapauksen avulla demonstroidaan automaattisen testauksen hyötyä syklisissä testeissä. Kuvassa 41 on esitettynä testissä tuotettu ohjaussignaali. Myös tämä signaali on tallennettu testauksen aikana kuvassa 38 näkyneestä log2 lohkosta.



Kuva 41. Testitapaus 2:n testisekvenssi.

Testitapaus 3: Ajaminen mäkeä ylös suurella lastilla ensin kiihdyttäen ja sitten tasaisella nopeudella

Tässä testissä testataan ohjausjärjestelmän toimintaa, kun tarvittava vääntö lisääntyy, mutta kaasun arvo pysyy samana. Tällöin moottorin tulisi nostaa kierroksia vastaamaan lisääntyvää momentin tarvetta.

5.3. Testaus

Testaus ja testien toistaminen on testitapausten määritysten jälkeen helppoa. GIMsim Test Manageriin ladataan testit, jonka jälkeen halutut testit suoritetaan ja analysoidaan nappia painamalla. Visualisoinnin avulla voitiin seurata testauksen etenemistä, mutta tämä ei ole välttämätöntä. Visualisoinnin avulla voidaan lisäksi todeta sekvenssien toimivuus helpommin kuin pelkästään tutkimalla testeistä saatuja signaaleja.

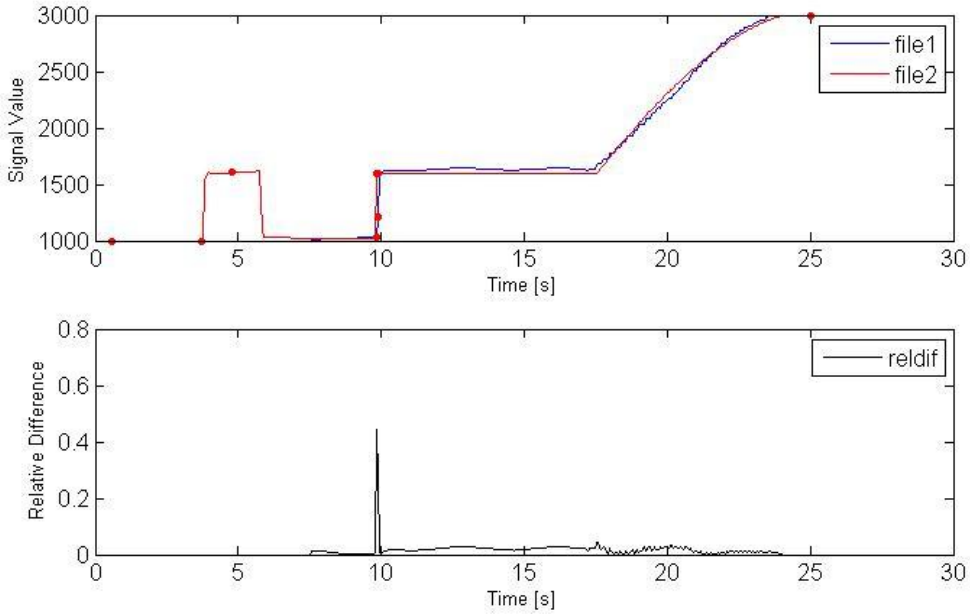
Kun yksi testi oli suoritettu, se analysoitiin välittömästi. Analysointi suoritettiin Test Analyzer ohjelmalla, joka vastaa määrittelyyn perustuvan vertailusignaalin generoinnista ja signaalien vertailusta. Ohjelmalle annettiin TestLinkiin tallennetut ja analysoinnissa käytettävät toleranssit, joiden perusteella ohjelma päättelee onko testissä saadut tulokset hyväksyttävissä rajoissa. Analysoinnissa käytettäväksi ajan toleranssiksi määriteltiin 0.5s, jonka on todettu olevan sopiva, poistamaan nopeiden liikkeiden aiheuttamat suuret mutta lyhytkestoiset virheet signaalien välillä. Virheen toleranssiksi valittiin 0.1, jonka avulla saadaan pienimmätkin yhtä kestoiset virheet havaittua. Analysoinnin tulokset lähetettiin TestLinkiin, josta niiden tuloksia pystyttiin tarkastelemaan.

Kaikkien kolmen testien suoritukseen kului aikaa yhteensä noin 10 minuuttia. Tästä ajasta testien alustuksiin kului noin 5 minuuttia.

5.4. Testauksen tulokset ja huomiot

Testien tulokset näkyvät TestLinkissä, josta niitä voidaan käydä tutkimassa. TestLinkistä saadaan helposti kokonaiskuva testauksen etenemisestä ja testitapauksista, joista löytyi virheitä.

Testitapauksessa 1 ajettiin kaasupolkimelle ramppivaste. Kuvassa 42 on esitettynä testissä saadut tulokset. Ylempi kuvaaja esittää kierrosluvun käyttäytymistä ja siinä punaisella kuvattu viiva on testissä saatu tulos ja sininen viiva on ohjausjärjestelmän määrittelyn perusteella generoitu vertailusignaali. Alemmassa kuvaajassa on esitettynä näiden kahden signaalin suhteellinen erotus.



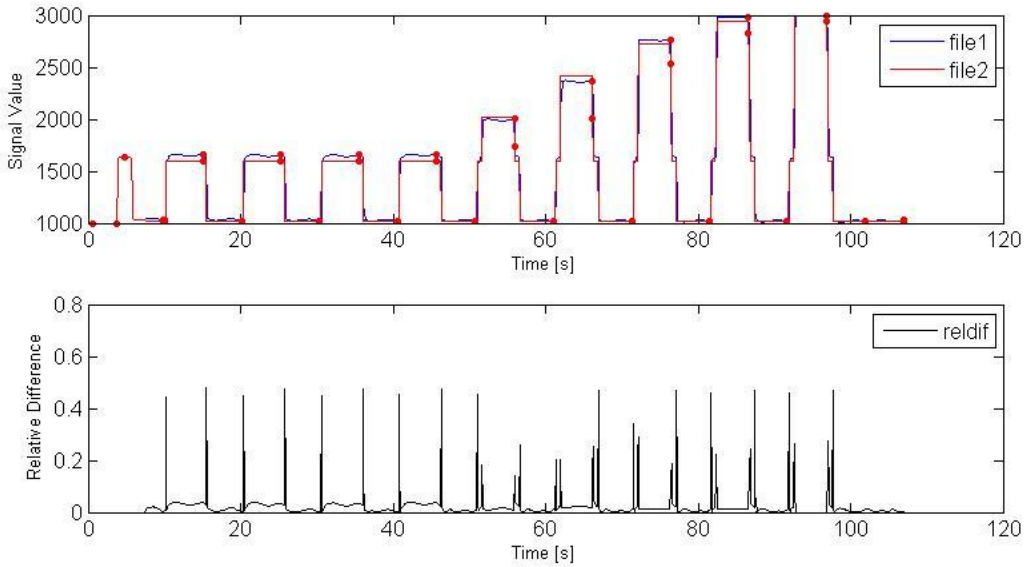
Kuva 42. Testitapauksesta 1 saatu kierroslukuvaste, jossa on esitetty testissä sekä analysoinnissa käytetyt signaalit ja niiden suhteellinen erotus.

Kuvaajasta voidaan huomata, etteivät viivat kulje kokoajan täysin päällekkäin, vaan simulaattorilta mitatussa signaalissa on pientä vaihtelua. Signaalit kulkevat kuitenkin tarpeeksi lähekkäin, joten testi voidaan määritellä läpäistyksi. 3-6 sekunnin kohdalla oleva hyppäys, johtuu koneen käynnistämisestä johtuvasta kierroslukujen tarpeen noususta. Sitä ei ole mallinnettu ohjausjärjestelmän määrittelyyn, jonka takia sitä ei myöskään oteta huomioon analysoinnissa.

10 sekunnin kohdalla oleva piikki alemmassa kuvaajassa on hyvä esimerkki suhteellisen erotuksen huonoista puolista analysoinnissa. Se ei ota huomioon nopeiden liikkeiden aiheuttamia lyhytkestoisia, mutta suuria virheitä. Tästä päästään kuitenkin eroon käyttämällä analysoinnissa filtteriä, jolloin lyhytkestoisia virheitä ei oteta

huomioon päätettäessä testin läpäisystä. Analysoinnin tuloksena testi arvioidaan siis läpäistyksi.

Testitapauksessa 2 ajettiin konetta eteen sykleittäin eri kaasun arvoilla. Kuvassa 43 on esitettynä testissä saadut tulokset. Signaalien värit tarkoittavat samaa kuin testitapauksessa 1.

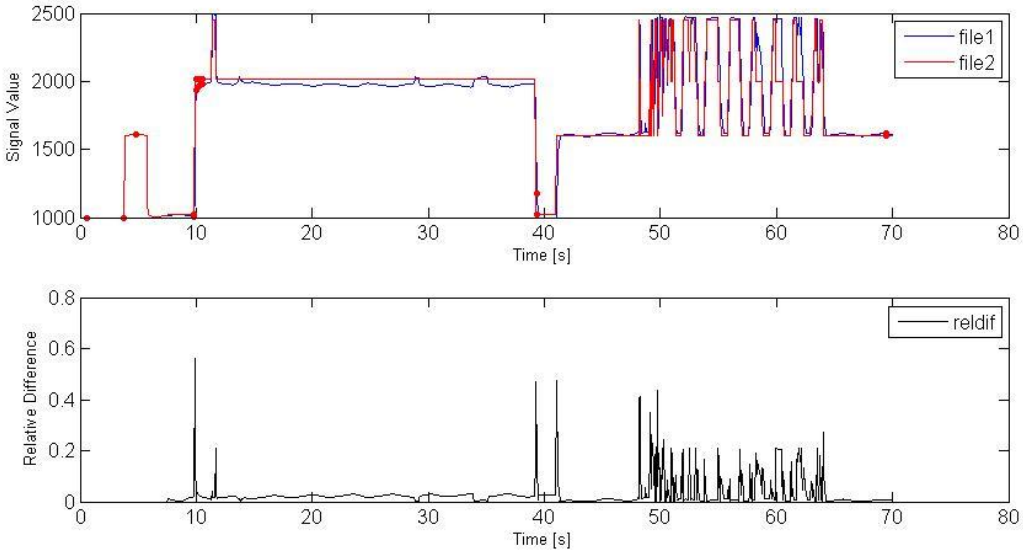


Kuva 43. Testitapauksesta 2 saatu kierroslukuvaste, jossa on esitetty testissä sekä analysoinnissa käytetyt signaalit ja niiden suhteellinen erotus.

Kuvaajasta nähdään että nopeiden muutosten kohdissa näkyy tässäkin testissä piikkejä, jotka suodatetaan pois käyttämällä filtteriä. Lisäksi jokaisen kiihdytyksen huipulla näkyy pientä eroavaisuutta, mutta erot ovat tarpeeksi pieniä mahtuakseen toleranssiin. Näin ollen analysoinnin tulos on hyväksytty. Saadut tulokset eivät kuvaa ramppivastetta yhtä tehokkaasti kuin testitapaus 1:ssa esitetyt tulokset. Kuitenkin voidaan huomata, kuinka ylemmän kuvaajan huiput vastaavat testitapauksessa 1 saatuja tuloksia samoilla kaasun arvoilla, jotka esitettiin kuvissa 40 ja 41.

Testin tarkoituksena oli havainnollistaa kuinka automaattinen testaus helpottaa syklisien testien tekoa. Vastaavan testin suorittaminen manuaalisesti olisi puuduttavaa toistoa ja samanlaisen tarkkuuden saavuttaminen lähes mahdotonta.

Testitapauksessa 3. ajettiin konetta mäkeä ylös ensin kiihdyttäen ja sitten tasaisella kaasulla. Kuvassa 44 on esitettynä testissä saadut tulokset. Signaalien värit tarkoittavat samaa kuin testitapauksessa 1.



Kuva 44. Testitapauksesta 3 saatu kierroslukuvaste, jossa on esitetty testissä sekä analysoinnissa käytetyt signaalit ja niiden suhteellinen erotus.

Kuvaajassa näkyy kaksi vaihetta. ensimmäisen 40 sekunnin aikana kone ajaa itsensä oikeaan asemaan, niin että se on kohtisuorassa mäkeen, Tämän jälkeen aloitetaan varsinainen testi. Mäkeä ajettaessa koneen kierrokset vaihtelevat 1600 ja 2400 kierroksen välissä, joka aiheutuu pyörien lipsumisesta ylämäessä. Alemmassa kuvaajassa tämä näkyy nopeina lyhytkestoisina vaihteluina. Kaikki nopeat vaihtelut kuitenkin mahtuvat filtterin arvon sisään, jolloin testi arvioidaan läpäistyksi.

Testien suoritukseen kului tässä testauksessa 10 minuuttia. Testaukseen käytettävää aikaa voidaan lyhentää muuttamalla prosessin sekvenssiä niin, että jos peräkkäisissä testeissä käytetään samoja malleja, niin niitä ei ladata joka kerralla uudelleen. Pelkkä simulointimallin ja signaalitallennuksen alustus riittäisivät.

Testitapausten luominen onnistui helposti käyttämällä vanhoja testejä uusien pohjana. Täysin uusien testitapausten luominen vaatii vielä paljon ulkoa muistamista kuten mallitiedostojen nimien muistamista ja tallennettavien signaalien nimien muistamista. Tätä saataisiin parannettua mahdollistamalla testien tietojen päivittäminen ja luominen Test Managerin kautta. Tällöin voitaisiin esimerkiksi testeissä tallennetavat signaalit valita signaalilistasta, ilman ulkoa muistamisen tarvetta. Vielä parempi vaihtoehto olisi sisällyttää tämä interaktiivisuus TestLinkiin, mutta se vaatisi huomattavaa muokkausta kyseiseen ohjelmistoon.

6. JOHTOPÄÄTÖKSET JA YHTEENVETO

Työkoneiden ohjausjärjestelmien kehitys on viime vuosina kiihtynyt ja järjestelmistä on tullut yhä monimutkaisempia. Tämän seurauksena myös niiden testaaminen on vaikeutunut. Apua tähän ongelmaan haetaan kehittämällä ohjausjärjestelmien automaattista testausta. Monet valmistajat lupaavat jo järjestelmiä, joiden avulla on mahdollista automatisoida ohjausjärjestelmien testausta, mutta TINAT- ja GIM-projekteissa saatujen kokemusten perusteella ne ovat kuitenkin rajoittuneita monelta osin. Tässä työssä tutkittiin ja perehdyttiin työkoneiden ohjausjärjestelmien automaattiseen testaukseen sekä esiteltiin automaattisen testauksen konsepti ja siihen liittyviä ohjelmistoja. Erityisesti perehdyttiin testausprosessiin, testitapausten luomiseen ja testauksen hallintaan. Työssä esiteltiin automaattisen testauksen konseptin toteutuksessa käytetty GIMsim Test Manager ohjelmisto, joka hallinnoi testausta.

Konsepti rakennettiin alustaksi, jotta valmistajien olisi helpompi hahmottaa automaattisen testauksen vaatimat osa-alueet. Konseptissa kuvattuja osa-alueita voidaan kehittää valmistajan tarpeiden mukaan, niin että syntynyt järjestelmä vastaa asiakaan vaatimuksia. Esimerkiksi jos valmistaja keskittyy ohjausjärjestelmän integraatiovaiheen testaukseen, jolloin yksittäiset ohjausyksiköt tulevat valmiina paketteina, voidaan keskittyä koko ohjausjärjestelmän määrityksiin perustuvaan funktionaaliseen testaukseen. Varsinaista ohjausyksiköiden sisäistä toimintaa ei tarvitse enää tällöin miettiä vaan riittää, että testataan ohjausyksiköiden yhteistoimintaa.

Konseptin mukaista toteutusta testattiin Avant-pienkuormaajan simulaattorilla. Siinä on useita reaaliaikakoneita, jotka yhdessä simuloivat pienkuormaajan toimintaa. Tällöin todettiin, että automaattinen testausjärjestelmä helpottaa työtä ja mahdollistaa tarkkojen signaalien antamisen testattavalle ohjausjärjestelmälle, joka ei ole aina ihmisen suorittamissa testeissä mahdollista. Kolmen testitapauksen avulla kuvattiin näitä järjestelmän tarjoamia mahdollisuuksia, kuten testitapausten tarkkaa määrittämistä ja suorittamista sekä toistuvien testien suorittamista.

Avant-pienkuormaajalla suoritettujen testien perusteella voidaan todeta järjestelmä toimivaksi. Se nopeutti testien suorittamista ja paransi testauksen tarkkuutta. Se myös mahdollisti testien nopean uusimisen tarpeen vaatiessa. Järjestelmää rakennettaessa oli otettu huomioon tietojen eriyttäminen, jonka ansiosta samojen testien uudelleen käyttäminen on helpompaa. Testauksessa löydettyjä virheitä pystyttiin korjaamaan simulointimalleja päivittämään ilman, että itse testitapauksiin tarvitsi puuttua.

Koska testien suorittaminen ja siihen liittyvä automatisointi on jo melko kehittynyttä, tulisi jatkossa kehityksen painopistettä siirtää automaattiseen testitapausten generointiin ja tulosten analysointiin. Tällöin testitapausten luominen helpottuisi ja

ohjausjärjestelmien määritysten muuttuessa uusien testitapausten generointi olisi nopeaa. Tämä mahdollistaa ohjausjärjestelmän nopeamman ja tarkemman testauksen kehitysprosessin aikaisemmissa vaiheissa, joka puolestaan voisi merkittävästi nopeuttaa uusien ominaisuuksien tuomista ohjausjärjestelmiin.

Automaattisella tulosten analysoinnilla helpotettaisiin kehittäjien työtä, kun kaikkia testejä ei tarvitsisi itse tarkistaa. Osittain tämä on jo mahdollista asettamalla tietylle testitapaukselle sopivia analysointimenetelmiä ja siihen liittyviä parametreja. Ongelma aiheutuu kuitenkin siitä, miten tiedetään että kyseinen menetelmä sopii kyseiselle testitapaukselle. Tulisikin siis kehittää menetelmiä, joilla tunnistetaan sopivat analysointimenetelmät ja parametrit kullekin testitapaukselle tai jopa jokaiselle signaalille kerrallaan.

Ohjausjärjestelmien automaattinen testaus nopeuttaa testaamista ja kehitystyötä, jonka takia yhä useammat yritykset ovat kiinnostuneita aiheesta. Monissa yrityksissä on myös omia projekteja testauksen kehittämiseksi ja voidaan olla varmoja, että automaattinen testaus on tullut jäädäkseen ohjausjärjestelmien testaukseen.

7. LÄHDELUETTELO

[1] FIMA - Forum for Intelligent Machines. [WWW]. [Viitattu: 14. 10 2009]. http://www.hermia.fi/fima/.

[2] Navet, N. & Simonot-Lion, F. Automotive Embedded Systems Handbook. 2008, CRC Press. 478 p.

[3] Hyvönen, M., Multanen, P., Mantere, P., Kolu, A., Vallas, A., Alanen, J. & Rantanen, S. Towards automatic testing of control systems for intelligent mobile machines. The 11th Scandinavian International Conference on Fluid Power, SICFP'09, June 2-4, 2009, Linköping, Sweden.

[4] ASAM-Association for Standardisation of Automation and Measuring Systems. [WWW]. [Viitattu: 25. 1 2010]. http://www.asam.net/.

[5] HIL- laitteisto- ja -palveluntarjonta. 2008. Tampere, FIMA. Julkaisematon esiselvitysraportti

[6] Holger, K., Lamberg, K. & Leinfellner, R. Automated Real-Time Testing of Electronic Control Unit. In-Vehicle Software & Hardware Systems. World Congress, Detroit, Michigan.2007. SAE International.

[7] Automaatiosuunnittelun prosessimalli - Yhteiset käsitteet verkottuneen suunnittelun perustana [WWW]. Suomen Automaatioseura ry. [Viitattu: 22.2.2010]. Saatavissa: www.automaatioseura.fi/ANTI-2.pdf

[8] Black, R. Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional. John Wiley & Sons, 2007. 384 p

[9] Broekman, B. & Notenboom, E. Testing embedded Software. 2003. Addison-Wesley Longman Publishing Co. 368 p

[10] ASAM AE HIL API. AE Hardware-in-the-Loop API. 2009. ASAM HIL Workgroup. 14 p.

[11] Myers, G.J. The Art of Software Testing. Hoboken, New Jersey. 2004, John Wiley & Sons. 252p.

[12] Berger, A.S. Embedded Systems Design: An Introduction to Processes, Tools, and Techniques. Lawrence, Kansas. 2002, CMP Books. 237.

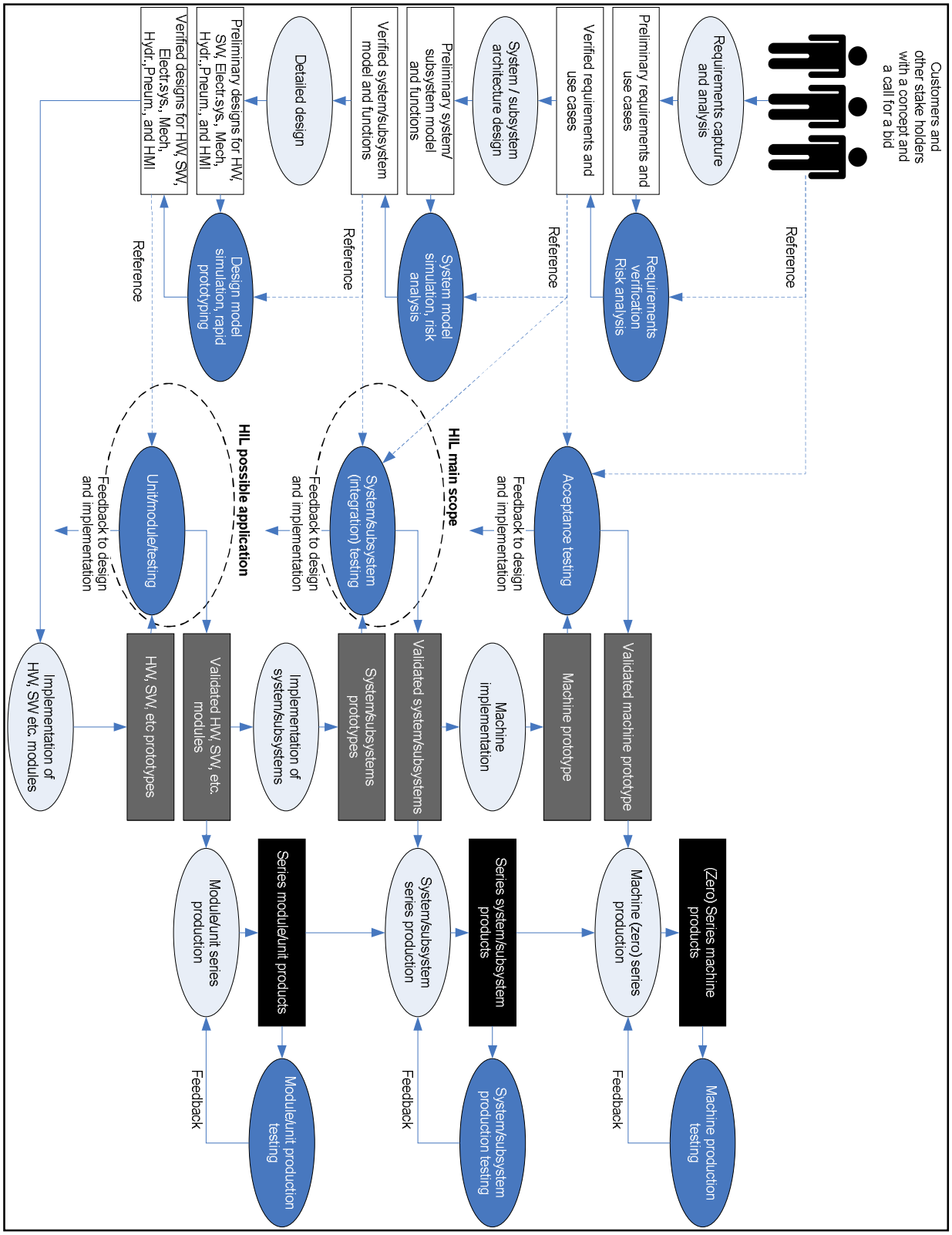
[13] Woit, D.M. Operational Profile Specification, Test Case Generation, and Reliability Estimation for Modules. Thesis. Kingston, Ontario, Kanada. Queen's University, Computing and Information Science 1994. 73p.

[14] TestLink - open source Test management. [WWW]. [Viitattu: 22.9.2009]. http://www.teamst.org/.

[15] Chilkat Software, Chilkat XML C++ Library. [WWW] [Viitattu: 28.5.2009]. http://www.chilkatsoft.com/XML-Library.asp.

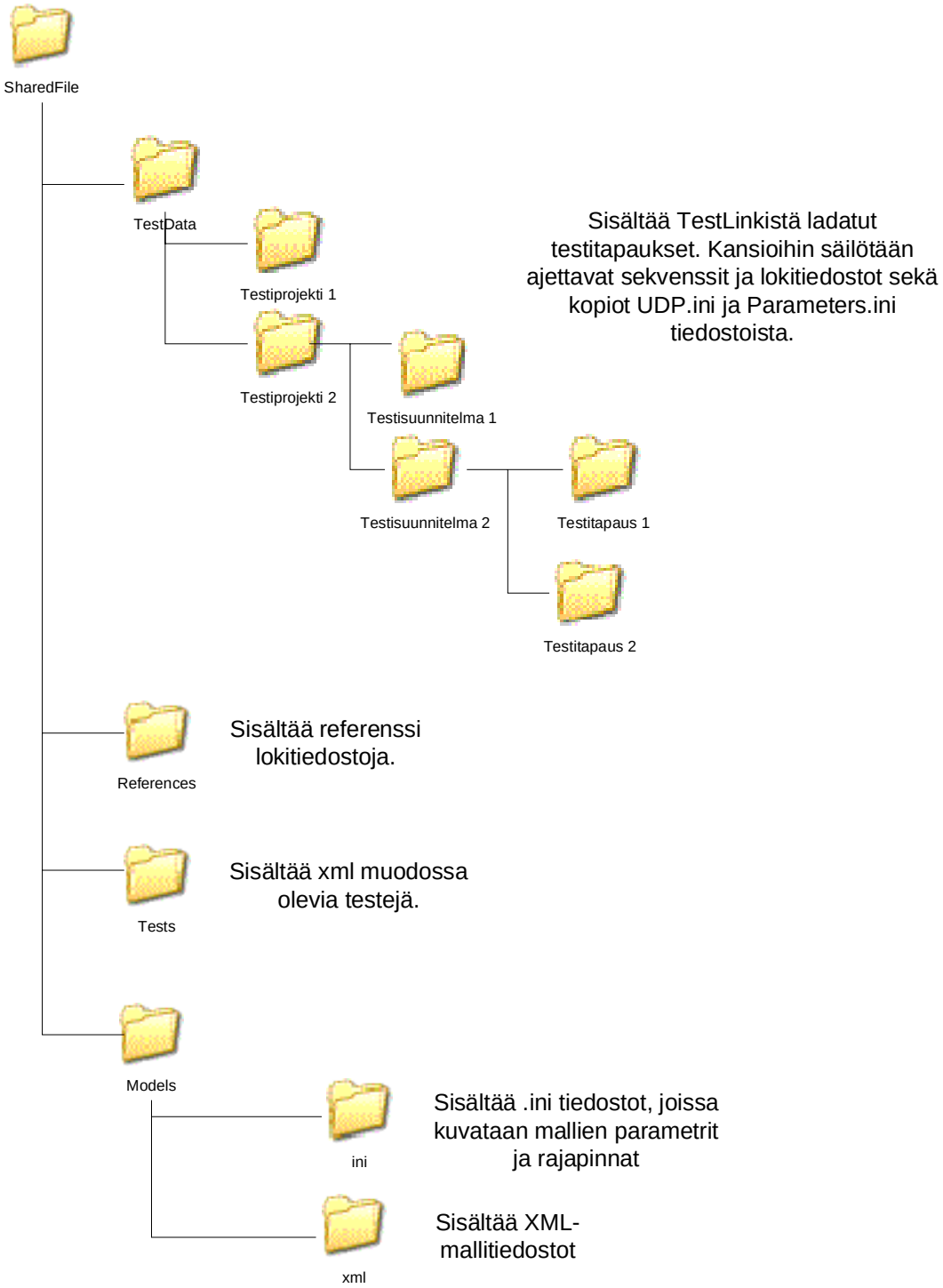
[16] Kidd, E. XML-RPC for C and C++. [WWW]. [Viitattu: 30.9.2009]. http://xmlrpc-c.sourceforge.net/.

Liite 1: Tripla V-malli [3]



Liite 2: TestLink ja GIMsim Test Manager ohjelmistojen käyttämä kansiorakenne

Mallien määrittelyyn käytetyt xml tiedosto, ohjausjärjestelmän kuvaus, makrot ja muut testien tiedot ovat tallennettuna kansiorakenteeseen, jaetulla verkkolevyllä. Kansiorakenteen tulee olla määrätyn kaltainen, jotta GIMsim Test Manager, osaisi etsiä tiedostoja oikeista paikoista.



Liite 3: XML- mallitiedosto

Useita malleja voidaan littää kerralla osaksi testiä, käyttämällä XML- mallitiedostoa. Siinä on kerrottuna mallien sijainti ja niiden tarvitsemat parametrit. XML- tiedosto on alla esitetyn kaltainen.

```
<?xml version="1.0" encoding="utf-8" ?>

<ModelList>
  <Model>
    <Name>liebherr_ctrl_w</Name>
    <Ip>192.168.0.9</Ip>
    <Port>22222</Port>
    <SimTime>-1.000000</SimTime>
    <StepTime>0.003000</StepTime>
    <Path>\\192.168.0.55\TestLinkShared\Models</Path>
    <Decimation>1</Decimation>
    <RefDecimation>1</RefDecimation>
    <ReferenceScopeList>
      <Scope>Receive from 25002/Unpack/s1</Scope>
      <Scope>Receive from 25002/Unpack/s2</Scope>
      <Scope>Receive from 25002/Unpack/s3</Scope>
      <Scope>Receive from 25002/Unpack/s4</Scope>
      <Scope>Receive from 25002/Unpack/s5</Scope>
      <Scope>Receive from 25002/Unpack/s6</Scope>
      <Scope>Receive from 25002/Unpack/s7</Scope>
      <Scope>Receive from 25002/Unpack/s8</Scope>
      <Scope>Receive from 25002/Unpack/s9</Scope>
    </ReferenceScopeList>
  </Model>
  <Model>
    <Name>liebherr_mech</Name>
    <Ip>192.168.0.10</Ip>
    <Port>22222</Port>
    <SimTime>-1.000000</SimTime>
    <StepTime>0.003000</StepTime>
    <Path>\\192.168.0.55\TestLinkShared\Models</Path>
    <Decimation>1</Decimation>
    <RefDecimation>1</RefDecimation>
    <ReferenceScopeList />
  </Model>
  <UDPIni>\\192.168.0.55\TestLinkShared\Models\liebherr_udp.ini</UDPIni>
   <ParametersIni>\\192.168.0.55\TestLinkShared\Models\liebherr_parameters.ini
</ParametersIni>
</ModelList>
```

Tiedostoon on tallennettu mallien tiedot sekä UDP.ini ja parameter.ini tiedostojen sijainnit. Malliin liittyviä tietoja ovat:

- Mallin nimi

- xPC Target koneen IP osoite, johon malli ladataan
- xPC Target koneen portin numero
- Simulointiaika, jossa -1 tarkoittaa jatkuvaa simulointia
- Askelaika
- Polku, jossa mallitiedosto sijaitsee
- Signaalien tallentamisen taajuus
- Referenssisignaalien tallentamisen taajuus
- Referenssisignaalit

Liite 4: Lokitiedoston rakenne

Lokitiedosto koostuu alussa olevasta otsikko osasta sekä data osasta. Alun otsikko osiossa on kerrottuna testiin liittyviä tietoja, joita Test Analyser käyttää analysoidessa ja visualisoidessa dataa. SequenceFile osoittaa sekvenssitiedoston, jota käytettiin testin ajamiseen. TestCaseID, TestPlanID, BuildID ovat TestLinkin testitapaukselle antamia tunnistustietoja. Niiden avulla pystytään analysaattorin antama tulos palauttamaan oikeaan testitapaukseen. DevKey on myös TestLink:n tunnistustieto, joka osoittaa kuka testin on suorittanut. ControlModel osoittaa ohjausjärjestelmän kuvaukseen, jonka avulla testin dataa voidaan analysoida.

LOG_START osoittaa kohdan, josta alkaa varsinainen testin data osa. Sen jälkeisellä rivillä on kirjoitettuna tallennettujen signaalien nimet erotettuna tabulaattorilla. Tämä jälkeen jokaisella rivillä on yhden ajanhetken arvot kullekin signaalille. Ensimmäisessä sarakkeessa on jokaisessa tiedostossa aika. Alla on esimerkki lokitiedoston rakenteesta.

```
SequenceFile=\\192.168.0.55\TestLinkShared\TestData\Liebherr_06-2009\Plan_liebherr\164_Vinssi_&_teleskooppi\PROG.txt
TestCaseID=164
TestPlanID=163
BuildID=19
DevKey=1b677f2f9310bd82ef6b7fe938773791
ControlModel=\\192.168.0.55\TestLinkShared\TestData\ControlModel\liebherr.m
LOG_START
Time        verify  lokkaus1/boom_angle  lokkaus1/boom_length
0.000000    0.000000        0.000000        10.500000
0.012000    0.000000        0.000000        10.500000
0.024000    0.000000        0.000000        10.500000
0.036000    0.000000        0.000000        10.500000
0.048000    0.000000        0.000000        10.500000
```

Liite 5: GimSim Test Managerin käyttötapaukset

Liitteessä on kuvattu Test Managerin käyttötapaukset sekä TestLinkin kaksi käyttötapausta, testien kirjoittaminen TestLinkiin ja tulosten tarkastelu TestLinkissä. Nämä ylimääräistet käyttötapaukset kuvattiin koska ne liittyvät läheisesti Test Managerin toimintaan.

1. Testien kirjoittaminen TestLinkiin

Testitapausten kirjoittaminen TestLinkiin aloitetaan luomalla uusi testitapaus halutun testausprojektin alle. TestLinkin testitapauksessa on kolme osiota, joista kahta käytetään testien kuvaamiseen. Summary osioon voidaan kuvata käytettävä XML- mallitiedosto, loki filtterit, Control model, loki decimaatio, analysoinnin toleranssit sekä määrätään, luodaanko FromFile sekvenssi vai ei.

Steps osioon kuvataan ajettava sekvenssi. Sekvenssi voi olla joko FromFile sekvenssi tai Sequence Generator sekvenssi. FromFile sekvenssi määritellään joko ajettavana tiedostona tai testinä, josta ajettava tiedosto otetaan.

Testiin voidaan myös ladata liitetiedostoja. Model, Control model ja FromFile sekvenssit voidaan ladata erikseen testin liitetiedostoksi. Jos kyseiset tiedostot on määritelty useammassa paikassa, käytetään Summary ja Steps osioissa määriteltyjä tiedostoja.

Jotta testien suoritus olisi mahdollista, ne pitää lisätä testaussuunnitelmaan. Testaussuunnitelma luodaan erikseen ohjelmaikkunan oikeassa reunassa olevan linkin kautta. Testaussuunnitelmaan pitää myös luoda vähintään yksi build, joka vastaa suorituksista testattavien versioiden välillä. Kun testattava ohjelmaversio vaihtuu, tulisi luoda uusi build sitä varten. Kun testitapaus on määritelty ja lisätty testaussuunnitelmaan, se voidaan ladata GIMsim Test Manageriin ja suorittaa.

2. Tulosten tarkastelu TestLinkissä

Kun testitapaukset on onnistuneesti suoritettu Test Managerissa, voidaan testin tuloksia tarkastella TestLinkissä. Tämä tapahtuu avaamalla testaussuunnitelma, jossa testitapaus on suoritettu. Testitapauksen viimeisin suorituskerta näkyy näytöllä, kuten myös suorituksen analysoinnista saadut tulokset. Vihreä passed teksti merkkaa läpäistyä testiä ja punainen failed teksti merkkaa hylättyä testiä. Testin lisäkentässä näkyy myös mahdolliset hylkäyksen aiheuttaneet virheet.

3. KT1: Testitapauksen luominen ja poistaminen

Esiehdot: -

Kuvaus: Painetaan Add test case-nappia. Tällöin ilmestyy uusi ikkuna johon kirjoitetaan testitapauksen nimi. Painettaessa OK-nappia, luodaan uusi testitapaus ja palataan pääikkunaan. Painettaessa Cancel-nappia, palataan pääikkunaan lisäämättä testitapausta.

Poikkeukset: Testitapauksella on sama nimi kuin jo olemassa olevalla testitapauksella, jolloin ilmoitetaan virheestä ja palataan pääikkunaan lisäämättä testitapausta.
Lopputulos: Uusi testitapaus on lisätty testilistaan.

4. KT2: Mallien lisääminen, poistaminen ja muokkaaminen

Esiehdot: Ohjelmaan on lisättynä ainakin yksi testitapaus.
Kuvaus: Valitaan testitapaus jota halutaan muokata. Painetaan Add new model-nappia jolloin avautuu uusi ikkuna, johon lisätään mallin tiedot. Mallin tietoihin kuuluu mallin nimi, polku, IP-osoite, portti, simulointiaika, simulointiaskel, loki taajuus sekä normaalille että referenssi signaalien tallentamiseen. Use sequence length-valinnan avulla testin pituus määräytyy ladatun sekvenssin pituuden perusteella. Mallin nimi ja polku voidaan myös hakea painamalla "…"-nappia, jolloin avautuu Windowsin tiedoston avaus ikkuna. Painettaessa OK-nappia mallin tiedot tallentuvat. Painettaessa Cancel-nappia palataan pääikkunaan ilman että mallin tietoja tallennetaan. Myöhemmin mallin tietoja voidaan muokata, valitsemalla haluttu malli, ja painamalla Edit->Model properties. Tällöin aukeaa sama ikkuna kuin mallia lisättäessä. Mallin voidaan poistaa painamalla Remove model-nappia, mallin ollessa valittuna.
Poikkeukset: -
Lopputulos: Malli on lisätty, poistettu tai sen tietoja on muokattu.

5. KT3: Testien lataaminen TestLinkistä

Esiehdot: TestLink ohjelmistoon on lisätty testitapauksia ja ne on määritetyn mallin mukaisia.
Kuvaus: Painetaan valikosta Menu->Load from TestLink-nappia. Tällöin ladataan kaikkien projektien kaikki testisuunnitelmat, jota sisältävät testitapauksia, ovat aktiivisia ja niissä on vähintään yksi build.
Poikkeukset: TestLink serverille ei saada yhteyttä, jolloin annetaan virheilmoitus.
Lopputulos: TestLink ohjelmistossa olevat testitapaukset on lisätty Test Manageriin.

6. KT4: Testien tallentaminen ja lataaminen XML- tiedostoon

Esiehdot: Ohjelmaan on lisätty ainakin yksi testitapaus.
Kuvaus: Painetaan valikosta Menu->Save to XML-nappia, jolloin avautuu Windowsin tiedoston tallennus ikkuna. Oletuksena aukeaa kansio, joka on määritetty Init.xml-tiedostossa. Kirjoitetaan tiedostolle nimi ja painetaan OK-nappia, jolloin Test Managerissa olevat testitapaukset tallentuvat XML-tiedostoksi. Tiedoston pystyy avaamaa painamalla

Menu->Load from XML-nappia, jolloin avautuu Windowsin tiedoston avaus ikkuna. Valitaan haluttu tiedosto ja painetaan OK-nappia, jolloin tiedostossa olleet testitapaukset avautuvat Test Manageriin.

Poikkeukset: XML-tiedosto ei ole oikean muotoinen, jolloin tietoja ei voida lukea.

Lopputulos: Ohjelmassa olevat tiedot on tallennettu XML-tiedostoon tai ne on ladattu takaisin Test Manageriin.

7. KT5: Tallennettavien signaalien valitseminen ja poistaminen

Esiehdot: Testitapaus ja malli on valittuna.

Kuvaus: Painetaan valikosta Edit->Test properties-nappia. Tällöin avautuu uusi ikkuna, jossa on kolme välilehteä. Toisessa välilehdessä voidaan valita tallennettavia signaaleja ja kolmannessa referenssi signaaleja, jotka muutetaan testin päätyttyä FromFile-lohko hyväksymäksi tiedostoksi. Signaali valitaan ruksimalla sen vieressä oleva laatikko. Useampia signaaleja voidaan valita maalaamalla. Tallennettavia signaaleja voidaan myös valita filtterin avulla, jolloin kaikki filtterin nimen sisältämät signaalit tallennetaan automaattisesti. TestLinkissä olevat filtterin arvot näkyvät filtterilistalla.

Poikkeukset: Mallin signaalit on käännösvaiheessa piilotettu (Inline), jolloin signaaleja ei voida valita. Tällöin malli tulee kääntää uudestaan ja asettaa signaalit näkymään.

Lopputulos: Signaalit on valittu tallennettaviksi

8. KT6: Parametrien muokkaaminen

Esiehdot: Testitapaus ja malli on valittuna.

Kuvaus: Painetaan valikosta Edit->Test properties-nappia. Tällöin avautuu uusi ikkuna, jossa on kolme välilehteä. Ensimmäisellä välilehdellä näkyy parametrit, joita on mahdollista muuttaa. Kaksoisklikkaamalla parametria aukeaa uusi ikkuna, jossa näkyy parametrin arvo. Arvon tilalle voidaan kirjoittaa uusi arvo ja painettaessa OK-nappia se tallentuu muistiin.

Poikkeukset: Parametreja on liikaa tai liian vähän, jolloin annetaan virheilmoitus.

Lopputulos: Parametrin arvo on tallennettu muistiin, josta se ladataan testin suorituksen alkaessa malliin.

9. KT7: Testien suoritus

Esiehdot: Ohjelmaan on lisätty ainakin yksi testitapaus.

Kuvaus: Valitaan suoritettavat testitapaukset ruksimalla niiden vasemmalla puolella olevat ruudut. Kun halutut testitapaukset on valittu, niin painetaan Run tests-nappia. Tällöin aukeaa uusi ikkuna, jossa voidaan seurata testauksen etenemistä. Painetaan Start-nappia, jolloin testien

suoritus käynnistyy. Kun testien suoritus on loppunut, painetaan Stop-nappia, jolloin palataan pääikkunaan.

Poikkeukset: Testauksen voi pysäyttää Stop-napista, jolloin kysytään halutaanko testin aikana saadut lokitiedostot tallentaa.

Lopputulos: Testit on saatu suoritettua.

10. KT8: Sekvenssin valitseminen

Esiehdot: Testitapaus on valittuna

Kuvaus: Ajettava sekvenssi voidaan valita Edit->Test Files-valikosta. Tällöin aukeaa uusi ikkuna, jossa on kaksi sekvenssivaihtoehtoa. Sequence generator-laatikkoon voidaan kirjoittaa sillä ajettavan sekvenssin nimi. From File-laatikkoon voidaan taas kirjoittaa sen sekvenssin nimi.

Poikkeukset: -

Lopputulos: Testitapauksessa käytettävä sekvenssi on valittu.

11. KT9: XML- mallitiedoston luominen testitapauksesta

Esiehdot: Testitapaus on luotu ja valittuna

Kuvaus: Valitaan valikosta Menu->Create TestLink model file from existing test case. Tämän jälkeen ohjelma kysyy UDP.ini ja parameters.ini tiedostoja, jotka liitetään mallitiedostoon. Kun ne on annettu, painetaan Save to-nappia jolloin luodaan tiedosto ja tallennetaan se käyttäjän antamaan kansioon. Luotu mallitiedosto on kuvattu liitteessä 2.

Poikkeukset: -

Lopputulos: XML- mallitiedosto on luotu